src/farfan_pipeline/phases/Phase_one/signal_enrichment.py

```
"""
Phase 1 Signal Enrichment Module
================================

Comprehensive signal integration for Phase 1 with maximum value aggregation.
This module provides advanced signal-driven analysis and enrichment capabilities
throughout all Phase 1 subphases (SP0-SP15).

Features:
- Questionnaire-aware signal extraction and application
- Signal-driven semantic scoring for entities and relationships
- Pattern-based causal marker detection
- Indicator-weighted evidence scoring
- Signal-enhanced temporal analysis
- Quality metrics and coverage tracking

Author: F.A.R.F.A.N Pipeline Team
Version: 2.0.0 - Maximum Signal Aggregation
"""

from __future__ import annotations

import re
from dataclasses import dataclass, field
from typing import Any, Dict, List, Optional
from pathlib import Path

# Module-level constants for scoring and thresholds
DIMENSIONS_PER_POLICY_AREA = 6
BASE_ENTITY_IMPORTANCE = 0.3
PATTERN_WEIGHT_FACTOR = 0.1
INDICATOR_WEIGHT_FACTOR = 0.15
ENTITY_WEIGHT_FACTOR = 0.1
MAX_PATTERN_WEIGHT = 0.3
MAX_INDICATOR_WEIGHT = 0.25
MAX_ENTITY_WEIGHT = 0.15
MAX_IMPORTANCE_SCORE = 0.95

# Default causal patterns (module-level constant)
DEFAULT_CAUSAL_PATTERNS = [
    (r'\bcausa\w*\b', 'CAUSE', 0.8),
    (r'\befecto\w*\b', 'EFFECT', 0.8),
    (r'\b(?:por lo tanto|por ende|en consecuencia)\b', 'CONSEQUENCE', 0.7),
    (r'\b(?:debido a|a causa de|producto de)\b', 'CAUSE_LINK', 0.75),
    (r'\b(?:resulta en|conduce a|genera)\b', 'EFFECT_LINK', 0.75),
    (r'\b(?:si|cuando).*(?:entonces|luego)\b', 'CONDITIONAL', 0.65),
]

# Base temporal patterns (module-level constant)
BASE_TEMPORAL_PATTERNS = [
    (r'\b(20\d{2})\b', 'YEAR', 0.9),
    (r'\b(\d{1,2})[/-](\d{1,2})[/-](20\d{2})\b', 'DATE', 0.85),
```

```
(r'\b(enero|febrero|marzo|abril|mayo|junio|julio|agosto|septiembre|octubre|noviembre|dic
iembre)\s+(?:de\s+)?(20\d{2})\b', 'MONTH_YEAR', 0.8),
    (r'\b(corto|mediano|largo)\s+plazo\b', 'HORIZON', 0.75),
    (r'\bvigencia\s+(20\d{2})[-?](20\d{2})\b', 'PERIOD', 0.85),
]

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

# Signal infrastructure imports
try:
        from  cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_registry
import (
        create_signal_registry,
    )
    from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signals import (
        create_default_signal_pack,
    )
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_quality_metrics
import (
        compute_signal_quality_metrics,
        analyze_coverage_gaps,
    )
    SISAS_AVAILABLE = True
except ImportError as e:
    logger.warning(f"SISAS not available: {e}")
    SISAS_AVAILABLE = False


@dataclass
class SignalEnrichmentContext:
    """
    Context for signal-based enrichment operations.

    Attributes:
        signal_registry: Questionnaire signal registry
        signal_packs: Policy area signal packs
        quality_metrics: Signal quality metrics per PA
        coverage_analysis: Coverage gap analysis
        provenance: Signal application provenance tracking
    """
    signal_registry: Optional[Any] = None
    signal_packs: Dict[str, Any] = field(default_factory=dict)
    quality_metrics: Dict[str, Any] = field(default_factory=dict)
    coverage_analysis: Optional[Any] = None
    provenance: Dict[str, List[str]] = field(default_factory=dict)
```

```
    def track_signal_application(self, chunk_id: str, signal_type: str, source: str) ->
None:
        """Track signal application for provenance."""
        if chunk_id not in self.provenance:
            self.provenance[chunk_id] = []
        self.provenance[chunk_id].append(f"{signal_type}:{source}")


class SignalEnricher:
    """
    Advanced signal enrichment engine for Phase 1.
    Provides comprehensive signal-based analysis across all subphases.
    """

    def __init__(self, questionnaire_path: Optional[Path] = None):
        """
        Initialize signal enricher.

        Args:
            questionnaire_path: Path to questionnaire JSON for signal extraction
        """
        self.context = SignalEnrichmentContext()
        self.questionnaire_path = questionnaire_path
        self._initialized = False

        if SISAS_AVAILABLE and questionnaire_path and questionnaire_path.exists():
            try:
                self._initialize_signal_registry(questionnaire_path)
            except Exception as e:
                logger.warning(f"Signal registry initialization failed: {e}")

    def _initialize_signal_registry(self, questionnaire_path: Path) -> None:
        """Initialize signal registry from questionnaire."""
        if not SISAS_AVAILABLE:
                        logger.warning("SISAS  not  available,  skipping  signal  registry
initialization")
            return

        try:
            # Load questionnaire and create signal registry
            # Note: create_signal_registry expects a loaded questionnaire object
            # For now, we'll use a fallback approach with default signal packs
            # A future enhancement would properly load the questionnaire first

            # Build signal packs for all policy areas
            for pa_num in range(1, 11):
                pa_id = f"PA{pa_num:02d}"
                try:
                    # Get signal pack - try from registry first, fallback to default
                    signal_pack = None
                                                if   self.context.signal_registry   and
hasattr(self.context.signal_registry, "get_signal_pack"):
                                                                        signal_pack   =
```

```
self.context.signal_registry.get_signal_pack(pa_id)

                    if signal_pack is None:
                        signal_pack = create_default_signal_pack(pa_id)

                    self.context.signal_packs[pa_id] = signal_pack

                    # Compute quality metrics
                    metrics = compute_signal_quality_metrics(signal_pack, pa_id)
                    self.context.quality_metrics[pa_id] = metrics

                    logger.info(
                        f"Loaded signal pack for {pa_id}: "
                        f"{len(signal_pack.patterns)} patterns, "
                        f"{len(signal_pack.indicators)} indicators, "
                        f"quality={metrics.coverage_tier}"
                    )
                except Exception as e:
                    logger.warning(f"Failed to load signal pack for {pa_id}: {e}")

            # Analyze coverage gaps
            if self.context.quality_metrics:
                try:
                    self.context.coverage_analysis = analyze_coverage_gaps(
                        list(self.context.quality_metrics.values())
                    )
                    logger.info(
                                                                f"Coverage   analysis:
{len(self.context.coverage_analysis.high_coverage_pas)} high, "
                        f"{len(self.context.coverage_analysis.low_coverage_pas)} low"
                    )
                except Exception as e:
                    logger.warning(f"Coverage analysis failed: {e}")

            self._initialized = True
            logger.info("Signal registry initialized successfully")

        except Exception as e:
            logger.error(f"Signal registry initialization error: {e}")
            self._initialized = False

    def enrich_entity_with_signals(
        self,
        entity_text: str,
        entity_type: str,
        policy_area: Optional[str] = None
    ) -> Dict[str, Any]:
        """
        Enrich entity with signal-based scoring.

        Args:
            entity_text: Entity text
            entity_type: Entity type (ACTOR, INDICADOR, TERRITORIO, etc.)
            policy_area: Target policy area (PA01-PA10)
```

```
        Returns:
            Dict with enrichment data including signal_tags, signal_scores, importance
        """
        enrichment = {
            'signal_tags': [entity_type],
            'signal_scores': {},
            'signal_importance': 0.5,  # Default baseline
            'matched_patterns': [],
            'matched_indicators': [],
            'matched_entities': [],
        }

        if not self._initialized or not policy_area:
            return enrichment

        entity_lower = entity_text.lower()
        signal_pack = self.context.signal_packs.get(policy_area)

        if not signal_pack:
            return enrichment

        # Match against signal patterns (use IGNORECASE flag, don't lowercase pattern)
        pattern_matches = 0
        for pattern in signal_pack.patterns:
            try:
                if re.search(pattern, entity_lower, re.IGNORECASE):
                    pattern_matches += 1
                    enrichment['matched_patterns'].append(pattern[:50])
                    enrichment['signal_tags'].append(f"PATTERN:{pattern[:20]}")
            except re.error:
                continue

        # Match against indicators
        indicator_matches = 0
        for indicator in signal_pack.indicators:
            if indicator.lower() in entity_lower:
                indicator_matches += 1
                enrichment['matched_indicators'].append(indicator)
                enrichment['signal_tags'].append(f"INDICATOR:{indicator[:20]}")

        # Match against entities
        entity_matches = 0
        for sig_entity in signal_pack.entities:
            if sig_entity.lower() in entity_lower or entity_lower in sig_entity.lower():
                entity_matches += 1
                enrichment['matched_entities'].append(sig_entity)
                enrichment['signal_tags'].append(f"ENTITY:{sig_entity[:20]}")

        # Calculate importance score based on matches using module constants
                        pattern_weight   =   min(MAX_PATTERN_WEIGHT,   pattern_matches   *
PATTERN_WEIGHT_FACTOR)
                    indicator_weight   =   min(MAX_INDICATOR_WEIGHT,   indicator_matches   *
INDICATOR_WEIGHT_FACTOR)
```

```
        entity_weight = min(MAX_ENTITY_WEIGHT, entity_matches * ENTITY_WEIGHT_FACTOR)

                        enrichment['signal_importance']   =   min(MAX_IMPORTANCE_SCORE,
BASE_ENTITY_IMPORTANCE + pattern_weight + indicator_weight + entity_weight)

        # Track signal scores per type
        if pattern_matches > 0:
               enrichment['signal_scores']['pattern_match'] = min(1.0, pattern_matches *
0.2)
        if indicator_matches > 0:
             enrichment['signal_scores']['indicator_match'] = min(1.0, indicator_matches
* 0.3)
        if entity_matches > 0:
                enrichment['signal_scores']['entity_match'] = min(1.0, entity_matches *
0.25)

        return enrichment

    def extract_causal_markers_with_signals(
        self,
        text: str,
        policy_area: str
    ) -> List[Dict[str, Any]]:
        """
        Extract causal markers using signal-driven pattern matching.

        Args:
            text: Text to analyze
            policy_area: Policy area for signal context

        Returns:
            List of causal markers with signal metadata
        """
        markers = []

        # Apply default patterns (use module-level constant)
        for pattern, marker_type, confidence in DEFAULT_CAUSAL_PATTERNS:
            for match in re.finditer(pattern, text, re.IGNORECASE):
                markers.append({
                    'text': match.group(0),
                    'type': marker_type,
                    'position': match.start(),
                    'confidence': confidence,
                    'source': 'default_pattern',
                })

        # Enhance with signal-based patterns if available
        if self._initialized and policy_area in self.context.signal_packs:
            signal_pack = self.context.signal_packs[policy_area]

            # Use signal patterns for causal detection
            for pattern in signal_pack.patterns:
                # Check if pattern might be causal-related
                pattern_lower = pattern.lower()
```

```
                    if any(kw in pattern_lower for kw in ['causa', 'efecto', 'impact',
'result', 'consecuen']):
                    try:
                        for match in re.finditer(pattern, text, re.IGNORECASE):
                            markers.append({
                                'text': match.group(0),
                                'type': 'SIGNAL_CAUSAL',
                                'position': match.start(),
                                      'confidence': 0.85,  # High confidence for signal
patterns
                                'source': f'signal_pattern:{pattern[:30]}',
                            })
                    except re.error:
                        continue

            # Use signal verbs for action-based causality
            for verb in signal_pack.verbs:
                verb_pattern = rf'\b{re.escape(verb)}\w*\b'
                try:
                    for match in re.finditer(verb_pattern, text, re.IGNORECASE):
                        markers.append({
                            'text': match.group(0),
                            'type': 'ACTION_VERB',
                            'position': match.start(),
                            'confidence': 0.7,
                            'source': f'signal_verb:{verb}',
                        })
                except re.error:
                    continue

        # Sort by position and deduplicate overlaps
        markers.sort(key=lambda m: m['position'])
        deduplicated = []
        last_end = -1

        for marker in markers:
            start = marker['position']
            end = start + len(marker['text'])

            if start >= last_end:
                deduplicated.append(marker)
                last_end = end

        return deduplicated

    def score_argument_with_signals(
        self,
        argument_text: str,
        argument_type: str,
        policy_area: str
    ) -> Dict[str, Any]:
        """
        Score argument strength using signal-based indicators.
```

```
Args:
    argument_text: Argument text
    argument_type: Argument type (claim, evidence, warrant, etc.)
    policy_area: Policy area context

Returns:
    Scoring dict with signal-enhanced metrics
"""
score = {
    'base_score': 0.5,
    'signal_boost': 0.0,
    'final_score': 0.5,
    'confidence': 0.5,
    'supporting_signals': [],
}

if not self._initialized or policy_area not in self.context.signal_packs:
    return score

signal_pack = self.context.signal_packs[policy_area]
text_lower = argument_text.lower()

# Boost for indicator presence (strong evidence)
indicator_count = 0
for indicator in signal_pack.indicators:
    if indicator.lower() in text_lower:
        indicator_count += 1
        score['supporting_signals'].append(f"indicator:{indicator}")

if indicator_count > 0:
    score['signal_boost'] += min(0.3, indicator_count * 0.15)

# Boost for entity mentions (contextual grounding)
entity_count = 0
for entity in signal_pack.entities:
    if entity.lower() in text_lower or text_lower in entity.lower():
        entity_count += 1
        score['supporting_signals'].append(f"entity:{entity}")

if entity_count > 0:
    score['signal_boost'] += min(0.15, entity_count * 0.1)

# Type-specific adjustments
if argument_type == 'evidence' and indicator_count > 0:
    score['confidence'] = min(0.9, 0.6 + indicator_count * 0.15)
elif argument_type == 'claim' and entity_count > 0:
    score['confidence'] = min(0.85, 0.5 + entity_count * 0.12)
else:
    score['confidence'] = 0.5 + score['signal_boost']

score['final_score'] = min(0.95, score['base_score'] + score['signal_boost'])

return score
```

```
    def extract_temporal_markers_with_signals(
        self,
        text: str,
        policy_area: str
    ) -> List[Dict[str, Any]]:
        """
        Extract temporal markers enhanced with signal patterns.

        Args:
            text: Text to analyze
            policy_area: Policy area context

        Returns:
            List of temporal markers with signal enrichment
        """
        markers = []

        # Use module-level BASE_TEMPORAL_PATTERNS constant
        for pattern, marker_type, confidence in BASE_TEMPORAL_PATTERNS:
            for match in re.finditer(pattern, text, re.IGNORECASE):
                markers.append({
                    'text': match.group(0),
                    'type': marker_type,
                    'confidence': confidence,
                    'source': 'base_temporal',
                })

        # Enhance with signal patterns
        if self._initialized and policy_area in self.context.signal_packs:
            signal_pack = self.context.signal_packs[policy_area]

            # Look for temporal patterns in signal catalog
            for pattern in signal_pack.patterns:
                pattern_lower = pattern.lower()
                       if any(kw in pattern_lower for kw in ['año', 'plazo', 'fecha',
'periodo', 'vigencia', 'temporal']):
                    try:
                        for match in re.finditer(pattern, text, re.IGNORECASE):
                            markers.append({
                                'text': match.group(0),
                                'type': 'SIGNAL_TEMPORAL',
                                'confidence': 0.82,
                                'source': f'signal:{pattern[:30]}',
                            })
                    except re.error:
                        continue

        return markers

    def compute_signal_coverage_metrics(
        self,
        chunks: List[Any]
    ) -> Dict[str, Any]:
        """
```

```
        Compute comprehensive signal coverage metrics for chunk set.

        Args:
            chunks: List of chunks to analyze

        Returns:
            Coverage metrics dict
        """
        metrics = {
            'total_chunks': len(chunks),
            'chunks_with_signals': 0,
            'avg_signal_tags_per_chunk': 0.0,
            'avg_signal_score': 0.0,
            'signal_density_by_pa': {},
            'signal_diversity': 0.0,
            'coverage_completeness': 0.0,
            'quality_tier': 'UNKNOWN',
        }

        if not chunks:
            return metrics

        all_signal_tags = set()
        total_signal_tags = 0
        total_signal_score = 0.0
        pa_signals = {}

        for chunk in chunks:
            signal_tags = getattr(chunk, 'signal_tags', [])
            signal_scores = getattr(chunk, 'signal_scores', {})

            if signal_tags:
                metrics['chunks_with_signals'] += 1
                total_signal_tags += len(signal_tags)
                all_signal_tags.update(signal_tags)

            if signal_scores:
                chunk_avg_score = sum(signal_scores.values()) / len(signal_scores)
                total_signal_score += chunk_avg_score

            # Track by policy area
            pa = getattr(chunk, 'policy_area_id', 'UNKNOWN')
            if pa not in pa_signals:
                pa_signals[pa] = {'count': 0, 'tags': set()}
            pa_signals[pa]['count'] += len(signal_tags)
            pa_signals[pa]['tags'].update(signal_tags)

        # Compute averages
        if metrics['chunks_with_signals'] > 0:
            metrics['avg_signal_tags_per_chunk'] = total_signal_tags / len(chunks)
                                metrics['avg_signal_score']   =   total_signal_score   /
metrics['chunks_with_signals']

        # Signal diversity (unique tags / total tags)
```

```
        if total_signal_tags > 0:
            metrics['signal_diversity'] = len(all_signal_tags) / total_signal_tags

        # Coverage completeness
        metrics['coverage_completeness'] = metrics['chunks_with_signals'] / len(chunks)

        # PA-level density
        for pa, data in pa_signals.items():
            metrics['signal_density_by_pa'][pa] = {
                'total_signals': data['count'],
                'unique_signals': len(data['tags']),
                'avg_per_chunk': data['count'] / DIMENSIONS_PER_POLICY_AREA,
            }

        # Quality tier classification
                                if    metrics['coverage_completeness']    >=    0.95    and
metrics['avg_signal_tags_per_chunk'] >= 5:
            metrics['quality_tier'] = 'EXCELLENT'
                               elif    metrics['coverage_completeness']    >=    0.85    and
metrics['avg_signal_tags_per_chunk'] >= 3:
            metrics['quality_tier'] = 'GOOD'
        elif metrics['coverage_completeness'] >= 0.70:
            metrics['quality_tier'] = 'ADEQUATE'
        else:
            metrics['quality_tier'] = 'SPARSE'

        return metrics

    def get_provenance_report(self) -> Dict[str, Any]:
        """
        Generate signal application provenance report.

        Returns:
            Provenance report with detailed tracking
        """
        return {
            'initialized': self._initialized,
            'signal_packs_loaded': list(self.context.signal_packs.keys()),
            'quality_metrics_available': list(self.context.quality_metrics.keys()),
            'coverage_analysis': self.context.coverage_analysis is not None,
                               'total_signal_applications':  sum(len(apps)  for  apps  in
self.context.provenance.values()),
            'chunks_enriched': len(self.context.provenance),
            'provenance_details': dict(self.context.provenance),
        }


def create_signal_enricher(questionnaire_path: Optional[Path] = None) -> SignalEnricher:
    """
    Factory function to create signal enricher instance.

    Args:
        questionnaire_path: Optional path to questionnaire for signal extraction
```

```
Returns:
    Configured SignalEnricher instance
"""
return SignalEnricher(questionnaire_path=questionnaire_path)
```

```
src/farfan_pipeline/phases/Phase_one/structural.py

"""
Structural normalization with policy-awareness.

Segments documents into policy-aware units.
"""

from typing import Any

# Provide calibrated_method stub if not available
try:
     from cross_cutting_infrastructure.capaz_calibration_parmetrization.decorators import
calibrated_method
except ImportError:
    # Stub decorator that does nothing
    def calibrated_method(name: str):
        def decorator(func):
            return func
        return decorator


class StructuralNormalizer:
    """Policy-aware structural normalizer."""


@calibrated_method("farfan_core.processing.spc_ingestion.structural.StructuralNormalizer
.normalize")
    def normalize(self, raw_objects: dict[str, Any]) -> dict[str, Any]:
        """
        Normalize document structure with policy awareness.

        Args:
            raw_objects: Raw parsed objects

        Returns:
            Policy graph with structured sections
        """
        policy_graph = {
            "sections": [],
            "policy_units": [],
            "axes": [],
            "programs": [],
            "projects": [],
            "years": [],
            "territories": [],
        }

        # Extract sections from pages
        for page in raw_objects.get("pages", []):
            text = page.get("text", "")

            # Detect policy units
            policy_units = self._detect_policy_units(text)
```

```
            policy_graph["policy_units"].extend(policy_units)

            # Create section
            section = {
                "text": text,
                "page": page.get("page_num"),
                "title": self._extract_title(text),
                "area": None,
                "eje": None,
            }
            policy_graph["sections"].append(section)

        # Extract axes, programs, projects
        for unit in policy_graph["policy_units"]:
            if unit["type"] == "eje":
                policy_graph["axes"].append(unit["name"])
            elif unit["type"] == "programa":
                policy_graph["programs"].append(unit["name"])
            elif unit["type"] == "proyecto":
                policy_graph["projects"].append(unit["name"])

        return policy_graph


@calibrated_method("farfan_core.processing.spc_ingestion.structural.StructuralNormalizer
._detect_policy_units")
    def _detect_policy_units(self, text: str) -> list[dict[str, Any]]:
        """Detect policy units in text."""
        units = []

        # Simple keyword-based detection
        keywords = {
            "eje": ["eje", "pilar"],
            "programa": ["programa"],
            "proyecto": ["proyecto"],
            "meta": ["meta"],
            "indicador": ["indicador"],
        }

        for unit_type, keywords_list in keywords.items():
            for keyword in keywords_list:
                if keyword.lower() in text.lower():
                    units.append({
                        "type": unit_type,
                        "name": f"{keyword.capitalize()} detected",
                    })

        return units


@calibrated_method("farfan_core.processing.spc_ingestion.structural.StructuralNormalizer
._extract_title")
    def _extract_title(self, text: str) -> str:
        """Extract title from text."""
```

```
# Simple: first line or first N characters
lines = text.split("\n")
if lines:
    return lines[0][:100]
return ""
```

src/farfan_pipeline/phases/Phase_three/__init__.py

```
"""Phase 3 - Scoring Module

This module handles transformation of Phase 2 execution results
into scored results ready for Phase 4 aggregation.

Key Functions:
- Extract validation scores from Phase 2 evidence
- Transform MicroQuestionRun to ScoredMicroQuestion
- Map policy areas and dimensions for aggregation
- Validate input counts, evidence presence, score bounds
"""

from farfan_pipeline.phases.Phase_three.validation import (
    VALID_QUALITY_LEVELS,
    ValidationCounters,
    validate_micro_results_input,
    validate_and_clamp_score,
    validate_quality_level,
    validate_evidence_presence,
)

__all__ = [
    "VALID_QUALITY_LEVELS",
    "ValidationCounters",
    "validate_micro_results_input",
    "validate_and_clamp_score",
    "validate_quality_level",
    "validate_evidence_presence",
]
```

```
src/farfan_pipeline/phases/Phase_three/scoring.py

"""Phase 3 Scoring Implementation

Transforms Phase 2 micro-question execution results into scored results
ready for Phase 4 aggregation. Extracts scores from EvidenceNexus outputs
(completeness, overall_confidence) and maps to Phase 4 format.

Key Functions:
- extract_score_from_nexus: Get overall_confidence from EvidenceNexus result
- map_completeness_to_quality: Map completeness enum to quality_level
- transform_micro_result_to_scored: Convert MicroQuestionRun to ScoredMicroQuestion

NOTE: Phase 2 uses EvidenceNexus which returns:
- overall_confidence (0.0-1.0) ? score
- completeness (complete/partial/insufficient/not_applicable) ? quality_level
- validation dict (passed/errors/warnings) - no score field unless validation fails
"""

from __future__ import annotations

import logging
from typing import Any

logger = logging.getLogger(__name__)

__all__ = [
    "extract_score_from_nexus",
    "map_completeness_to_quality",
    "extract_score_from_evidence",
    "extract_quality_level",
    "transform_micro_result_to_scored",
]


def extract_score_from_nexus(result_data: dict[str, Any]) -> float:
    """Extract numeric score from Phase 2 EvidenceNexus result.

    Phase 2 uses EvidenceNexus which computes overall_confidence (0.0-1.0).
    This is the primary scoring metric from the evidence graph.

    Args:
        result_data: Full result dict from Phase 2 executor

    Returns:
        overall_confidence score (0.0-1.0), defaults to 0.0 if not found
    """
    # Primary: Get overall_confidence from nexus result
    confidence = result_data.get("overall_confidence")
    if confidence is not None:
        try:
            return float(confidence)
        except (TypeError, ValueError):
            logger.warning(f"Invalid overall_confidence type: {type(confidence)}")
```

```
    # Fallback: Check validation.score (only set when validation fails with score_zero
policy)
    validation = result_data.get("validation", {})
    score = validation.get("score")
    if score is not None:
        try:
            return float(score)
        except (TypeError, ValueError):
            logger.warning(f"Invalid validation score type: {type(score)}")

    # Fallback: Use mean confidence from evidence elements
    evidence = result_data.get("evidence", {})
    if isinstance(evidence, dict):
        conf_scores = evidence.get("confidence_scores", {})
        mean_conf = conf_scores.get("mean")
        if mean_conf is not None:
            try:
                return float(mean_conf)
            except (TypeError, ValueError):
                pass

    # Default: 0.0
    return 0.0


def map_completeness_to_quality(completeness: str | None) -> str:
    """Map EvidenceNexus completeness enum to quality level.

    Completeness values from EvidenceNexus:
    - "complete" ? "EXCELENTE"
    - "partial" ? "ACEPTABLE"
    - "insufficient" ? "INSUFICIENTE"
    - "not_applicable" ? "NO_APLICABLE"

    Args:
        completeness: Completeness enum value from nexus result

    Returns:
        Quality level string for Phase 4 aggregation
    """
    if not completeness:
        return "INSUFICIENTE"

    completeness_lower = completeness.lower()

    mapping = {
        "complete": "EXCELENTE",
        "partial": "ACEPTABLE",
        "insufficient": "INSUFICIENTE",
        "not_applicable": "NO_APLICABLE",
    }

    return mapping.get(completeness_lower, "INSUFICIENTE")
```

```
def extract_score_from_evidence(evidence: dict[str, Any] | None) -> float:
    """DEPRECATED: Extract score from evidence dict (legacy).

    Use extract_score_from_nexus() instead. This function is kept for
    backward compatibility but doesn't align with EvidenceNexus architecture.

    Args:
        evidence: Evidence dict from MicroQuestionRun (may be None)

    Returns:
        Extracted score (0.0-1.0), defaults to 0.0 if not found
    """
    if not evidence:
        return 0.0

    # Try validation.score (only set when validation fails)
    validation = evidence.get("validation", {})
    score = validation.get("score")

    if score is not None:
        try:
            return float(score)
        except (TypeError, ValueError):
            logger.warning(f"Invalid score type in validation: {type(score)}")

    # Try confidence_scores.mean as fallback
    if isinstance(evidence, dict):
        conf_scores = evidence.get("confidence_scores", {})
        mean_conf = conf_scores.get("mean")
        if mean_conf is not None:
            try:
                return float(mean_conf)
            except (TypeError, ValueError):
                pass

    return 0.0


def extract_quality_level(evidence: dict[str, Any] | None, completeness: str | None =
None) -> str:
    """Extract quality level from Phase 2 result.

    Args:
        evidence: Evidence dict from MicroQuestionRun (may be None)
        completeness: Completeness enum from nexus result (preferred)

    Returns:
        Quality level string, defaults to "INSUFICIENTE" if not found
    """
    # Primary: Map completeness if available
    if completeness:
        return map_completeness_to_quality(completeness)
```

```
    # Fallback: Check validation.quality_level (only set when validation fails)
    if evidence:
        validation = evidence.get("validation", {})
        quality = validation.get("quality_level")

        if quality is not None:
            return str(quality)

    return "INSUFICIENTE"


def transform_micro_result_to_scored(
    micro_result: Any,
) -> dict[str, Any]:
    """DEPRECATED: Transform MicroQuestionRun to ScoredMicroQuestion dict.

    NOTE: This function is deprecated and not used by the orchestrator.
    The orchestrator._score_micro_results_async() performs the transformation
    directly using extract_score_from_nexus() and map_completeness_to_quality().

    This function is kept for backward compatibility and testing purposes only.

    Args:
        micro_result: MicroQuestionRun from Phase 2

    Returns:
        Dict ready for ScoredMicroQuestion dataclass construction

    Deprecated:
        Use orchestrator._score_micro_results_async() for production.
    """
    question_id = getattr(micro_result, "question_id", None)
    question_global = getattr(micro_result, "question_global", 0)
    base_slot = getattr(micro_result, "base_slot", "UNKNOWN")
    metadata = getattr(micro_result, "metadata", {})
    evidence_obj = getattr(micro_result, "evidence", None)
    error = getattr(micro_result, "error", None)

    # Extract evidence dict (Evidence dataclass or dict)
    if hasattr(evidence_obj, "__dict__"):
        evidence = evidence_obj.__dict__
    elif isinstance(evidence_obj, dict):
        evidence = evidence_obj
    else:
        evidence = {}

    # DEPRECATED: Extract score and quality from evidence (legacy fallback)
    # Production code should use extract_score_from_nexus(metadata) instead
    score = extract_score_from_evidence(evidence)
    quality_level = extract_quality_level(evidence)

    # Build scored result dict
    scored = {
```

```
    "question_id": question_id,
    "question_global": question_global,
    "base_slot": base_slot,
    "score": score,
    "normalized_score": score,  # Already normalized 0.0-1.0
    "quality_level": quality_level,
    "evidence": evidence_obj,  # Keep original Evidence object
    "scoring_details": {
        "source": "legacy_fallback",
        "method": "extract_from_evidence",
    },
    "metadata": metadata,
    "error": error,
}

return scored
```

```
src/farfan_pipeline/phases/Phase_three/signal_enriched_scoring.py

"""Phase 3 Signal-Enriched Scoring Module

Extends Phase 3 scoring with signal-based enhancements for increased rigor
and determinism. Provides signal-driven threshold adjustments and quality
mapping with full provenance tracking.

Enhancement Value:
- Signal-based threshold adaptation based on question complexity
- Signal-driven quality level validation
- Enhanced scoring provenance with signal metadata
- Deterministic, byte-reproducible scoring with signal context

Integration: Used by orchestrator._score_micro_results_async() to enhance
basic scoring with signal intelligence.

Author: F.A.R.F.A.N Pipeline Team
Version: 1.0.0
"""

from __future__ import annotations

import logging
from typing import Any, TYPE_CHECKING

if TYPE_CHECKING:
    try:
         from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_registry
import (
            QuestionnaireSignalRegistry,
        )
    except ImportError:
        QuestionnaireSignalRegistry = Any  # type: ignore

logger = logging.getLogger(__name__)

# Quality level constants
QUALITY_EXCELENTE = "EXCELENTE"
QUALITY_ACEPTABLE = "ACEPTABLE"
QUALITY_INSUFICIENTE = "INSUFICIENTE"
QUALITY_NO_APLICABLE = "NO_APLICABLE"
QUALITY_ERROR = "ERROR"

# Threshold adjustment constants
HIGH_PATTERN_THRESHOLD = 15  # Pattern count threshold for complexity
HIGH_INDICATOR_THRESHOLD = 10  # Indicator count threshold for specificity
PATTERN_COMPLEXITY_ADJUSTMENT = -0.05  # Lower threshold for complex questions
INDICATOR_SPECIFICITY_ADJUSTMENT = 0.03  # Raise threshold for specific questions
COMPLETE_EVIDENCE_ADJUSTMENT = 0.02  # Bonus for complete evidence

# Score thresholds for validation
HIGH_SCORE_THRESHOLD = 0.8
LOW_SCORE_THRESHOLD = 0.3
```

```
__all__ = [
    "SignalEnrichedScorer",
    "get_signal_adjusted_threshold",
    "get_signal_quality_validation",
    # Constants
    "QUALITY_EXCELENTE",
    "QUALITY_ACEPTABLE",
    "QUALITY_INSUFICIENTE",
    "QUALITY_NO_APLICABLE",
    "QUALITY_ERROR",
]


class SignalEnrichedScorer:
    """Signal-enriched scorer for Phase 3 with deterministic enhancements.

    Adds signal intelligence to basic Phase 3 scoring without changing
    core scoring logic. All enhancements are additive and optional.

    Attributes:
        signal_registry: Optional signal registry for signal-driven scoring
        enable_threshold_adjustment: Enable signal-based threshold tuning
        enable_quality_validation: Enable signal-based quality validation
    """

    def __init__(
        self,
        signal_registry: QuestionnaireSignalRegistry | None = None,
        enable_threshold_adjustment: bool = True,
        enable_quality_validation: bool = True,
    ) -> None:
        """Initialize signal-enriched scorer.

        Args:
            signal_registry: Optional signal registry for signal access
            enable_threshold_adjustment: Enable threshold adjustment feature
            enable_quality_validation: Enable quality validation feature
        """
        self.signal_registry = signal_registry
        self.enable_threshold_adjustment = enable_threshold_adjustment
        self.enable_quality_validation = enable_quality_validation

        logger.info(
            f"SignalEnrichedScorer initialized: "
            f"registry={'enabled' if signal_registry else 'disabled'}, "
            f"threshold_adj={enable_threshold_adjustment}, "
            f"quality_val={enable_quality_validation}"
        )

    def adjust_threshold_for_question(
        self,
        question_id: str,
        base_threshold: float,
```

```
        score: float,
        metadata: dict[str, Any],
    ) -> tuple[float, dict[str, Any]]:
        """Adjust scoring threshold based on signal-driven question complexity.

        Uses signal registry to determine question complexity and adjust
        thresholds accordingly. More complex questions get slightly lower
        thresholds to account for increased difficulty.

        Args:
            question_id: Question identifier (e.g., "Q001")
            base_threshold: Base threshold from scoring system
            score: Computed score for the question
            metadata: Question metadata dict

        Returns:
            Tuple of (adjusted_threshold, adjustment_details)
        """
        if not self.enable_threshold_adjustment or not self.signal_registry:
                            return  base_threshold,  {"adjustment":  "none",  "reason":
"disabled_or_no_registry"}

        adjustment_details: dict[str, Any] = {
            "base_threshold": base_threshold,
            "adjustments": [],
        }

        adjusted = base_threshold

        try:
            # Get micro answering signals for question
            signal_pack = self.signal_registry.get_micro_answering_signals(question_id)

            # Adjust based on pattern complexity (more patterns = more complex)
            pattern_count = len(getattr(signal_pack, 'patterns', []))
            if pattern_count > HIGH_PATTERN_THRESHOLD:
                adjustment = PATTERN_COMPLEXITY_ADJUSTMENT
                adjusted = max(0.3, adjusted + adjustment)
                adjustment_details["adjustments"].append({
                    "type": "high_pattern_complexity",
                    "pattern_count": pattern_count,
                    "adjustment": adjustment,
                })

            # Adjust based on indicator count (more indicators = more specific)
            indicator_count = len(getattr(signal_pack, 'indicators', []))
            if indicator_count > HIGH_INDICATOR_THRESHOLD:
                adjustment = INDICATOR_SPECIFICITY_ADJUSTMENT
                adjusted = min(0.9, adjusted + adjustment)
                adjustment_details["adjustments"].append({
                    "type": "high_indicator_specificity",
                    "indicator_count": indicator_count,
                    "adjustment": adjustment,
                })
```

```
        # Adjust based on evidence quality from metadata
        completeness = metadata.get("completeness", "").lower()
        if completeness == "complete":
            adjustment = COMPLETE_EVIDENCE_ADJUSTMENT
            adjusted = min(0.9, adjusted + adjustment)
            adjustment_details["adjustments"].append({
                "type": "complete_evidence",
                "completeness": completeness,
                "adjustment": adjustment,
            })

        adjustment_details["adjusted_threshold"] = adjusted
        adjustment_details["total_adjustment"] = adjusted - base_threshold

        logger.debug(
            f"Threshold adjusted for {question_id}: "
            f"{base_threshold:.3f} ? {adjusted:.3f} "
            f"(?={adjusted - base_threshold:+.3f})"
        )

    except Exception as e:
        logger.warning(
            f"Failed to adjust threshold for {question_id}: {e}. "
            f"Using base threshold {base_threshold:.3f}"
        )
        adjustment_details["error"] = str(e)
        adjusted = base_threshold

    return adjusted, adjustment_details

def validate_quality_level(
    self,
    question_id: str,
    quality_level: str,
    score: float,
    completeness: str | None,
) -> tuple[str, dict[str, Any]]:
    """Validate and potentially adjust quality level using signal intelligence.

    Uses signal-based heuristics to ensure quality level is consistent
    with score, completeness, and question characteristics.

    Args:
        question_id: Question identifier
        quality_level: Computed quality level from Phase 3
        score: Numeric score (0.0-1.0)
        completeness: Completeness enum from EvidenceNexus

    Returns:
        Tuple of (validated_quality_level, validation_details)
    """
    if not self.enable_quality_validation:
        return quality_level, {"validation": "disabled"}
```

```
        validation_details: dict[str, Any] = {
            "original_quality": quality_level,
            "score": score,
            "completeness": completeness,
            "checks": [],
        }

        validated = quality_level

        try:
            # Check 1: Score-quality consistency
             if score >= HIGH_SCORE_THRESHOLD and quality_level in [QUALITY_INSUFICIENTE,
QUALITY_NO_APLICABLE]:
                validation_details["checks"].append({
                    "check": "score_quality_consistency",
                    "issue": "high_score_low_quality",
                    "action": "promote_quality",
                })
                validated = QUALITY_ACEPTABLE  # Promote to at least ACEPTABLE
                logger.info(
                    f"Quality promoted for {question_id}: "
                    f"{quality_level} ? {validated} (high score {score:.3f})"
                )

            # Check 2: Completeness-quality alignment
            if completeness == "complete" and quality_level == QUALITY_INSUFICIENTE:
                validation_details["checks"].append({
                    "check": "completeness_quality_alignment",
                    "issue": "complete_evidence_low_quality",
                    "action": "promote_quality",
                })
                      validated = QUALITY_ACEPTABLE   # At least ACEPTABLE for complete
evidence
                logger.info(
                    f"Quality promoted for {question_id}: "
                    f"{quality_level} ? {validated} (complete evidence)"
                )

            # Check 3: Low score validation
            if score < LOW_SCORE_THRESHOLD and quality_level == QUALITY_EXCELENTE:
                validation_details["checks"].append({
                    "check": "low_score_validation",
                    "issue": "low_score_high_quality",
                    "action": "demote_quality",
                })
                validated = QUALITY_ACEPTABLE  # Demote to ACEPTABLE
                logger.info(
                    f"Quality demoted for {question_id}: "
                    f"{quality_level} ? {validated} (low score {score:.3f})"
                )

            validation_details["validated_quality"] = validated
            validation_details["adjusted"] = validated != quality_level
```

```
        except Exception as e:
            logger.warning(
                f"Failed to validate quality for {question_id}: {e}. "
                f"Using original quality {quality_level}"
            )
            validation_details["error"] = str(e)
            validated = quality_level

        return validated, validation_details

    def enrich_scoring_details(
        self,
        question_id: str,
        base_scoring_details: dict[str, Any],
        threshold_adjustment: dict[str, Any],
        quality_validation: dict[str, Any],
    ) -> dict[str, Any]:
        """Enrich scoring details with signal provenance.

        Adds signal-based metadata to scoring details for full transparency
        and reproducibility.

        Args:
            question_id: Question identifier
            base_scoring_details: Base scoring details from Phase 3
            threshold_adjustment: Threshold adjustment details
            quality_validation: Quality validation details

        Returns:
            Enriched scoring details dict
        """
        enriched = {
            **base_scoring_details,
            "signal_enrichment": {
                "enabled": True,
                "registry_available": self.signal_registry is not None,
                "threshold_adjustment": threshold_adjustment,
                "quality_validation": quality_validation,
            },
        }

        return enriched


def get_signal_adjusted_threshold(
    signal_registry: QuestionnaireSignalRegistry | None,
    question_id: str,
    base_threshold: float,
    score: float,
    metadata: dict[str, Any],
) -> tuple[float, dict[str, Any]]:
    """Convenience function for signal-based threshold adjustment.
```

```
    Creates a temporary SignalEnrichedScorer and returns adjusted threshold.

    Args:
        signal_registry: Signal registry instance (optional)
        question_id: Question identifier
        base_threshold: Base threshold value
        score: Computed score
        metadata: Question metadata

    Returns:
        Tuple of (adjusted_threshold, adjustment_details)
    """
    scorer = SignalEnrichedScorer(signal_registry=signal_registry)
    return scorer.adjust_threshold_for_question(
        question_id=question_id,
        base_threshold=base_threshold,
        score=score,
        metadata=metadata,
    )


def get_signal_quality_validation(
    question_id: str,
    quality_level: str,
    score: float,
    completeness: str | None,
) -> tuple[str, dict[str, Any]]:
    """Convenience function for signal-based quality validation.

    Creates a temporary SignalEnrichedScorer and returns validated quality.

    Args:
        question_id: Question identifier
        quality_level: Original quality level
        score: Computed score
        completeness: Completeness enum

    Returns:
        Tuple of (validated_quality, validation_details)
    """
    scorer = SignalEnrichedScorer()
    return scorer.validate_quality_level(
        question_id=question_id,
        quality_level=quality_level,
        score=score,
        completeness=completeness,
    )
```

```
src/farfan_pipeline/phases/Phase_three/validation.py

"""Phase 3 Validation Module

Provides strict validation for Phase 3 scoring pipeline to prevent:
- Missing or incomplete micro-question results
- Out-of-bounds score values
- Invalid quality level enums
- Missing or null evidence
- Silent score corruption
"""

from __future__ import annotations

import logging
from dataclasses import dataclass
from typing import Any

logger = logging.getLogger(__name__)

__all__ = [
    "VALID_QUALITY_LEVELS",
    "ValidationCounters",
    "validate_micro_results_input",
    "validate_and_clamp_score",
    "validate_quality_level",
    "validate_evidence_presence",
]


VALID_QUALITY_LEVELS    =    frozenset({"EXCELENTE",    "ACEPTABLE",    "INSUFICIENTE",
"NO_APLICABLE"})


@dataclass
class ValidationCounters:
    """Tracks validation failures during Phase 3 scoring."""
    total_questions: int = 0
    missing_evidence: int = 0
    out_of_bounds_scores: int = 0
    invalid_quality_levels: int = 0
    score_clamping_applied: int = 0
    quality_level_corrections: int = 0

    def log_summary(self) -> None:
        """Log validation summary."""
        logger.info(
            f"Phase 3 validation summary: total_questions={self.total_questions}, "
            f"missing_evidence={self.missing_evidence}, "
            f"out_of_bounds_scores={self.out_of_bounds_scores}, "
            f"invalid_quality_levels={self.invalid_quality_levels}, "
            f"score_clamping_applied={self.score_clamping_applied}, "
            f"quality_level_corrections={self.quality_level_corrections}"
        )
```

```
        if self.out_of_bounds_scores > 0:
            logger.warning(
                  f"Phase 3: {self.out_of_bounds_scores} scores were out of bounds [0.0,
1.0]"
            )

        if self.invalid_quality_levels > 0:
            logger.warning(
                f"Phase 3: {self.invalid_quality_levels} quality levels were invalid"
            )


def validate_micro_results_input(
    micro_results: list[Any],
    expected_count: int,
) -> None:
    """Validate micro-question results input before scoring.

    Args:
        micro_results: List of MicroQuestionRun objects from Phase 2
        expected_count: Expected number of questions (default: 305)

    Raises:
        ValueError: If input validation fails
    """
    if not micro_results:
        raise ValueError("Phase 3 input validation failed: micro_results list is empty")

    actual_count = len(micro_results)
    if actual_count != expected_count:
        logger.error(
            f"Phase 3 input validation failed: expected_count={expected_count}, "
            f"actual_count={actual_count}, difference={actual_count - expected_count}"
        )
        raise ValueError(
             f"Phase 3 input validation failed: Expected {expected_count} micro-question
"
            f"results but got {actual_count}"
        )

    logger.info(f"Phase 3 input validation passed: question_count={actual_count}")


def validate_evidence_presence(
    evidence: Any,
    question_id: str,
    question_global: int,
    counters: ValidationCounters,
) -> bool:
    """Validate that evidence is present and not null.

    Args:
        evidence: Evidence object from MicroQuestionRun
```

```
        question_id: Question identifier
        question_global: Global question number
        counters: Validation counters to update

    Returns:
        True if evidence is valid, False otherwise
    """
    if evidence is None:
        counters.missing_evidence += 1
        logger.error(
            f"Phase 3 evidence validation failed: question_id={question_id}, "
            f"question_global={question_global}, reason=evidence is None"
        )
        return False

    return True


def validate_and_clamp_score(
    score: float | None,
    question_id: str,
    question_global: int,
    counters: ValidationCounters,
) -> float:
    """Validate score is in [0.0, 1.0] range and clamp if needed.

    Args:
        score: Raw score value
        question_id: Question identifier
        question_global: Global question number
        counters: Validation counters to update

    Returns:
        Clamped score in [0.0, 1.0] range
    """
    if score is None:
        logger.warning(
            f"Phase 3 score validation: score is None, defaulting to 0.0, "
            f"question_id={question_id}, question_global={question_global}"
        )
        return 0.0

    try:
        score_float = float(score)
    except (TypeError, ValueError) as e:
        counters.out_of_bounds_scores += 1
        logger.error(
            f"Phase 3 score validation failed: unconvertible type, "
            f"question_id={question_id}, question_global={question_global}, "
                        f"score_type={type(score).__name__},  score_value={str(score)},
error={str(e)}"
        )
        return 0.0
```

```
    if score_float < 0.0 or score_float > 1.0:
        counters.out_of_bounds_scores += 1
        counters.score_clamping_applied += 1
        clamped = max(0.0, min(1.0, score_float))
        logger.warning(
            f"Phase 3 score clamping applied: question_id={question_id}, "
            f"question_global={question_global}, original_score={score_float}, "
            f"clamped_score={clamped}"
        )
        return clamped

    return score_float


def validate_quality_level(
    quality_level: str | None,
    question_id: str,
    question_global: int,
    counters: ValidationCounters,
) -> str:
    """Validate quality level is from valid enum set.

    Valid values: EXCELENTE, ACEPTABLE, INSUFICIENTE, NO_APLICABLE

    Args:
        quality_level: Quality level string
        question_id: Question identifier
        question_global: Global question number
        counters: Validation counters to update

    Returns:
        Validated quality level (corrected to INSUFICIENTE if invalid)
    """
    if quality_level is None:
        counters.invalid_quality_levels += 1
        counters.quality_level_corrections += 1
        logger.warning(
             f"Phase 3 quality level validation: None value, correcting to INSUFICIENTE,
"
            f"question_id={question_id}, question_global={question_global}"
        )
        return "INSUFICIENTE"

    quality_str = str(quality_level).strip()

    if quality_str not in VALID_QUALITY_LEVELS:
        counters.invalid_quality_levels += 1
        counters.quality_level_corrections += 1
        logger.error(
            f"Phase 3 quality level validation failed: question_id={question_id}, "
            f"question_global={question_global}, invalid_value={quality_str}, "
            f"valid_values={list(VALID_QUALITY_LEVELS)}, corrected_to=INSUFICIENTE"
        )
        return "INSUFICIENTE"
```

```
return quality_str
```

src/farfan_pipeline/phases/Phase_two/__init__.py

```
"""Orchestrator utilities with contract validation on import."""

from __future__ import annotations

from typing import TYPE_CHECKING

# NEW: Evidence processing - REAL PATH: canonic_phases.Phase_two.evidence_nexus
# Replaces evidence_assembler, evidence_validator, evidence_registry
from canonic_phases.Phase_two.evidence_nexus import (
    EvidenceNexus,
    EvidenceGraph,
    EvidenceNode,
    process_evidence,
)

# NEW: Narrative synthesis - REAL PATH: canonic_phases.Phase_two.carver
from canonic_phases.Phase_two.carver import (
    DoctoralCarverSynthesizer,
    CarverAnswer,
)

# Executor config - REAL PATH: canonic_phases.Phase_two.executor_config
from canonic_phases.Phase_two.executor_config import ExecutorConfig

__all__ = [
    # NEW: Evidence processing (EvidenceNexus)
    "EvidenceNexus",
    "EvidenceGraph",
    "EvidenceNode",
    "process_evidence",
    # NEW: Narrative synthesis (Carver)
    "DoctoralCarverSynthesizer",
    "CarverAnswer",
    # Orchestration core
    "Orchestrator",
    "MethodExecutor",
    "Evidence",
    "AbortSignal",
    "AbortRequested",
    "ResourceLimits",
    "PhaseInstrumentation",
    "PhaseResult",
    "MicroQuestionRun",
    "ScoredMicroQuestion",
    "ExecutorConfig",
]
```

```
src/farfan_pipeline/phases/Phase_two/arg_router.py

"""Argument routing with special routes, strict validation, and comprehensive metrics.

This module provides ExtendedArgRouter (and legacy ArgRouter for compatibility):
- 30+ special route handlers for commonly-called methods
- Strict validation (no silent parameter drops)
- **kwargs support for forward compatibility
- Full observability and metrics
- Base routing and validation utilities

Design Principles:
- Explicit route definitions for high-traffic methods
- Fail-fast on missing required arguments
- Fail-fast on unexpected arguments (unless **kwargs present)
- Full traceability of routing decisions
- Zero tolerance for silent parameter drops
"""

from __future__ import annotations

import inspect
import logging
import os
import random
import threading
from collections.abc import Iterable, Mapping, MutableMapping
from dataclasses import dataclass
from typing import (
    Any,
    Union,
    get_args,
    get_origin,
    get_type_hints,
)

try:
    import structlog  # type: ignore
except Exception:  # pragma: no cover
    structlog = None

std_logger = logging.getLogger(__name__)

if structlog is None:
    class _CompatLogger:
        def __init__(self, base: logging.Logger) -> None:
            self._base = base

        def debug(self, event: str, **kwargs: Any) -> None:
            self._base.debug("%s %s", event, kwargs if kwargs else "")

        def info(self, event: str, **kwargs: Any) -> None:
            self._base.info("%s %s", event, kwargs if kwargs else "")
```

```
        def warning(self, event: str, **kwargs: Any) -> None:
            self._base.warning("%s %s", event, kwargs if kwargs else "")

        def error(self, event: str, **kwargs: Any) -> None:
            self._base.error("%s %s", event, kwargs if kwargs else "")

    logger = _CompatLogger(std_logger)
else:
    logger = structlog.get_logger(__name__)

# Sentinel value for missing arguments
MISSING: object = object()


# ============================================================================
# Base Exceptions and Data Classes
# ============================================================================

class ArgRouterError(RuntimeError):
    """Base exception for routing and validation issues."""


class ArgumentValidationError(ArgRouterError):
    """Raised when the provided payload does not match the method signature."""

    def __init__(
        self,
        class_name: str,
        method_name: str,
        *,
        missing: Iterable[str] | None = None,
        unexpected: Iterable[str] | None = None,
        type_mismatches: Mapping[str, str] | None = None,
    ) -> None:
        self.class_name = class_name
        self.method_name = method_name
        self.missing = set(missing or ())
        self.unexpected = set(unexpected or ())
        self.type_mismatches = dict(type_mismatches or {})
        detail = []
        if self.missing:
            detail.append(f"missing={sorted(self.missing)}")
        if self.unexpected:
            detail.append(f"unexpected={sorted(self.unexpected)}")
        if self.type_mismatches:
            detail.append(f"type_mismatches={self.type_mismatches}")
        message = (
            f"Invalid payload for {class_name}.{method_name}"
            + (f" ({'; '.join(detail)})" if detail else "")
        )
        super().__init__(message)


@dataclass(frozen=True)
```

```
class _ParameterSpec:
    name: str
    kind: inspect._ParameterKind
    default: Any
    annotation: Any

    @property
    def required(self) -> bool:
        return self.default is MISSING


@dataclass(frozen=True)
class MethodSpec:
    class_name: str
    method_name: str
    positional: tuple[_ParameterSpec, ...]
    keyword_only: tuple[_ParameterSpec, ...]
    has_var_keyword: bool
    has_var_positional: bool

    @property
    def required_arguments(self) -> tuple[str, ...]:
        required = tuple(
            spec.name
            for spec in (*self.positional, *self.keyword_only)
            if spec.required
        )
        return required

    @property
    def accepted_arguments(self) -> tuple[str, ...]:
        accepted = tuple(spec.name for spec in (*self.positional, *self.keyword_only))
        return accepted


# ============================================================================
# Base ArgRouter (Legacy - use ExtendedArgRouter instead)
# ============================================================================

class ArgRouter:
    """Resolve method call payloads based on inspected signatures.

    .. note::
        ExtendedArgRouter is the recommended router to use directly.
        This base class is provided for backward compatibility.
    """

    def __init__(self, class_registry: Mapping[str, type]) -> None:
        self._class_registry = dict(class_registry)
        self._spec_cache: dict[tuple[str, str], MethodSpec] = {}
        self._lock = threading.RLock()

    def describe(self, class_name: str, method_name: str) -> MethodSpec:
        """Return the cached method specification, building it if necessary."""
```

```
    key = (class_name, method_name)
    with self._lock:
        if key not in self._spec_cache:
            self._spec_cache[key] = self._build_spec(class_name, method_name)
        return self._spec_cache[key]

def route(
    self,
    class_name: str,
    method_name: str,
    payload: MutableMapping[str, Any],
) -> tuple[tuple[Any, ...], dict[str, Any]]:
    """Validate and split a payload into positional and keyword arguments."""
    spec = self.describe(class_name, method_name)
    provided_keys = set(payload.keys())
    required = set(spec.required_arguments)
    accepted = set(spec.accepted_arguments)

    missing = required - provided_keys
    unexpected = provided_keys - accepted
    if unexpected and spec.has_var_keyword:
        unexpected = set()

    if missing or unexpected:
        raise ArgumentValidationError(
            class_name,
            method_name,
            missing=missing,
            unexpected=unexpected,
        )

    args: list[Any] = []
    kwargs: dict[str, Any] = {}
    type_mismatches: dict[str, str] = {}

    remaining = dict(payload)

    for param in spec.positional:
        if param.name not in remaining:
            if param.required:
                missing = {param.name}
                raise ArgumentValidationError(
                    class_name,
                    method_name,
                    missing=missing,
                )
            continue
        value = remaining.pop(param.name)
        if not self._matches_annotation(value, param.annotation):
            expected = self._describe_annotation(param.annotation)
            type_mismatches[param.name] = expected
        args.append(value)

    for param in spec.keyword_only:
```

```
            if param.name not in remaining:
                if param.required:
                    raise ArgumentValidationError(
                        class_name,
                        method_name,
                        missing={param.name},
                    )
                continue
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            kwargs[param.name] = value

        if spec.has_var_keyword and remaining:
            kwargs.update(remaining)
            remaining = {}

        if remaining:
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=set(remaining.keys()),
            )

        if type_mismatches:
            raise ArgumentValidationError(
                class_name,
                method_name,
                type_mismatches={
                                                name:  f"expected  {expected};  received
{type(payload[name]).__name__}"
                    for name, expected in type_mismatches.items()
                },
            )

        return tuple(args), kwargs

    def expected_arguments(self, class_name: str, method_name: str) -> tuple[str, ...]:
        spec = self.describe(class_name, method_name)
        return spec.accepted_arguments

    def _build_spec(self, class_name: str, method_name: str) -> MethodSpec:
        try:
            cls = self._class_registry[class_name]
        except KeyError as exc:  # pragma: no cover - defensive
            raise ArgRouterError(f"Unknown class '{class_name}'") from exc

        try:
            method = getattr(cls, method_name)
        except AttributeError as exc:
             raise ArgRouterError(f"Class '{class_name}' has no method '{method_name}'")
from exc
```

```
    signature = inspect.signature(method)
    try:
        type_hints = get_type_hints(method)
    except Exception:
        type_hints = {}
    positional: list[_ParameterSpec] = []
    keyword_only: list[_ParameterSpec] = []
    has_var_keyword = False
    has_var_positional = False

    for parameter in signature.parameters.values():
        if parameter.name == "self":
            continue
        default = (
            parameter.default
            if parameter.default is not inspect._empty
            else MISSING
        )
        annotation = type_hints.get(parameter.name, parameter.annotation)
        param_spec = _ParameterSpec(
            name=parameter.name,
            kind=parameter.kind,
            default=default,
            annotation=annotation,
        )
        if parameter.kind in (
            inspect.Parameter.POSITIONAL_ONLY,
            inspect.Parameter.POSITIONAL_OR_KEYWORD,
        ):
            positional.append(param_spec)
        elif parameter.kind is inspect.Parameter.KEYWORD_ONLY:
            keyword_only.append(param_spec)
        elif parameter.kind is inspect.Parameter.VAR_KEYWORD:
            has_var_keyword = True
        elif parameter.kind is inspect.Parameter.VAR_POSITIONAL:
            has_var_positional = True

    return MethodSpec(
        class_name=class_name,
        method_name=method_name,
        positional=tuple(positional),
        keyword_only=tuple(keyword_only),
        has_var_keyword=has_var_keyword,
        has_var_positional=has_var_positional,
    )

@staticmethod
def _matches_annotation(value: Any, annotation: Any) -> bool:
    if annotation in (inspect._empty, Any):
        return True
    origin = get_origin(annotation)
    if origin is None:
        if isinstance(annotation, type):
            return isinstance(value, annotation)
```

```
            return True
        args = get_args(annotation)
        if origin is tuple:
            if not isinstance(value, tuple):
                return False
            if not args:
                return True
            if len(args) == 2 and args[1] is Ellipsis:
                    return all(ArgRouter._matches_annotation(item, args[0]) for item in
value)
            if len(args) != len(value):
                return False
            return all(
                ArgRouter._matches_annotation(item, arg_type)
                for item, arg_type in zip(value, args, strict=False)
            )
        if origin in (list, list):
            if not isinstance(value, list):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (set, set):
            if not isinstance(value, set):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (dict, dict):
            if not isinstance(value, dict):
                return False
            if len(args) != 2:
                return True
            key_type, value_type = args
            return all(
                ArgRouter._matches_annotation(k, key_type)
                and ArgRouter._matches_annotation(v, value_type)
                for k, v in value.items()
            )
        if origin is Union:
            return any(ArgRouter._matches_annotation(value, arg) for arg in args)
        return True

    @staticmethod
    def _describe_annotation(annotation: Any) -> str:
        if annotation in (inspect._empty, Any):
            return "Any"
        origin = get_origin(annotation)
        if origin is None:
            if isinstance(annotation, type):
                return annotation.__name__
            return str(annotation)
        args = get_args(annotation)
        if origin is tuple:
```

```
                return f"Tuple[{', '.join(ArgRouter._describe_annotation(arg) for arg in
args)}]"
        if origin in (list, list):
                 return f"List[{ArgRouter._describe_annotation(args[0])}]" if args else
"List[Any]"
        if origin in (set, set):
                 return f"Set[{ArgRouter._describe_annotation(args[0])}]" if args else
"Set[Any]"
        if origin in (dict, dict):
            if len(args) == 2:
                return (
                    f"Dict[{ArgRouter._describe_annotation(args[0])}, "
                    f"{ArgRouter._describe_annotation(args[1])}]"
                )
            return "Dict[Any, Any]"
        if origin is Union:
            return " | ".join(ArgRouter._describe_annotation(arg) for arg in args)
        return str(annotation)


class PayloadDriftMonitor:
    """Sampling validator for ingress/egress payloads."""

    CRITICAL_KEYS = {
        "content": str,
        "pdq_context": (dict, type(None)),
    }

    def __init__(self, *, sample_rate: float, enabled: bool) -> None:
        self.sample_rate = max(0.0, min(sample_rate, 1.0))
        self.enabled = enabled and self.sample_rate > 0.0

    @classmethod
    def from_env(cls) -> PayloadDriftMonitor:
        enabled = os.getenv("ORCHESTRATOR_SAMPLING_VALIDATION", "").lower() in {
            "1",
            "true",
            "yes",
            "on",
        }
        try:
            sample_rate = float(os.getenv("ORCHESTRATOR_SAMPLING_RATE", "0.05"))
        except ValueError:
            sample_rate = 0.05
        return cls(sample_rate=sample_rate, enabled=enabled)

     def maybe_validate(self, payload: Mapping[str, Any], *, producer: str, consumer:
str) -> None:
        if not self.enabled:
            return
        if random.random() > self.sample_rate:
            return
        if not isinstance(payload, Mapping):
            return
```

```
        keys = set(payload.keys())
        if not keys.intersection(self.CRITICAL_KEYS):
            return

        missing = [key for key in self.CRITICAL_KEYS if key not in payload]
        type_mismatches = {
            key: self._expected_type_name(expected)
            for key, expected in self.CRITICAL_KEYS.items()
            if key in payload and not isinstance(payload[key], expected)
        }
        if missing or type_mismatches:
            std_logger.error(
                "Payload drift detected [%s -> %s]: missing=%s type_mismatches=%s",
                producer,
                consumer,
                missing,
                type_mismatches,
            )
        else:
            std_logger.debug(
                "Payload validation OK [%s -> %s]", producer, consumer
            )

    @staticmethod
    def _expected_type_name(expected: object) -> str:
        if isinstance(expected, tuple):
            return ", ".join(getattr(t, "__name__", str(t)) for t in expected)
        if hasattr(expected, "__name__"):
            return expected.__name__  # type: ignore[arg-type]
        return str(expected)


# ============================================================================
# Extended ArgRouter with Special Routes
# ============================================================================


@dataclass
class RoutingMetrics:
    """Metrics for monitoring routing behavior."""

    total_routes: int = 0
    special_routes_hit: int = 0
    default_routes_hit: int = 0
    validation_errors: int = 0
    silent_drops_prevented: int = 0


class ExtendedArgRouter(ArgRouter):
    """
    Extended argument router with special route handling.

    Extends base ArgRouter with:
    - 25+ special route definitions
```

```
- Strict validation (no silent drops)
- **kwargs awareness for forward compatibility
- Comprehensive metrics

Special Routes (?25):
1. _extract_quantitative_claims
2. _parse_number
3. _determine_semantic_role
4. _compile_pattern_registry
5. _analyze_temporal_coherence
6. _validate_evidence_chain
7. _calculate_confidence_score
8. _extract_indicators
9. _parse_temporal_reference
10. _determine_policy_area
11. _compile_regex_patterns
12. _analyze_source_reliability
13. _validate_numerical_consistency
14. _calculate_bayesian_update
15. _extract_entities
16. _parse_citation
17. _determine_validation_type
18. _compile_indicator_patterns
19. _analyze_coherence_score
20. _validate_threshold_compliance
21. _calculate_evidence_weight
22. _extract_temporal_markers
23. _parse_budget_allocation
24. _determine_risk_level
25. _compile_validation_rules
26. _analyze_stakeholder_impact
27. _validate_governance_structure
28. _calculate_alignment_score
29. _extract_constraint_declarations
30. _parse_implementation_timeline
"""

def __init__(self, class_registry: Mapping[str, type]) -> None:
    """
    Initialize extended router.

    Args:
        class_registry: Mapping of class names to class types
    """
    super().__init__(class_registry)
    self._special_routes = self._build_special_routes()
    self._metrics = RoutingMetrics()
    self._metrics_lock = threading.Lock()

    logger.info(
        "extended_arg_router_initialized",
        special_routes=len(self._special_routes),
        classes=len(class_registry),
    )
```

```
def _build_special_routes(self) -> dict[str, dict[str, Any]]:
    """
    Build special route definitions for commonly-called methods.

    Each route specifies:
    - required_args: List of required parameter names
    - optional_args: List of optional parameter names
    - accepts_kwargs: Whether method accepts **kwargs
    - description: Human-readable description

    Returns:
        Dict mapping method names to route specs
    """
    routes = {
        "_extract_quantitative_claims": {
            "required_args": ["content"],
            "optional_args": ["context", "thresholds", "patterns"],
            "accepts_kwargs": True,
            "description": "Extract quantitative claims from content",
        },
        "_parse_number": {
            "required_args": ["text"],
            "optional_args": ["locale", "unit_system"],
            "accepts_kwargs": True,
            "description": "Parse numerical value from text",
        },
        "_determine_semantic_role": {
            "required_args": ["text", "context"],
            "optional_args": ["role_taxonomy", "confidence_threshold"],
            "accepts_kwargs": True,
            "description": "Determine semantic role of text element",
        },
        "_compile_pattern_registry": {
            "required_args": ["patterns"],
            "optional_args": ["category", "flags"],
            "accepts_kwargs": False,
            "description": "Compile patterns into regex registry",
        },
        "_analyze_temporal_coherence": {
            "required_args": ["content"],
            "optional_args": ["temporal_patterns", "baseline_date"],
            "accepts_kwargs": True,
            "description": "Analyze temporal coherence of content",
        },
        "_validate_evidence_chain": {
            "required_args": ["claims", "evidence"],
            "optional_args": ["validation_rules", "min_confidence"],
            "accepts_kwargs": True,
            "description": "Validate evidence chain for claims",
        },
        "_calculate_confidence_score": {
            "required_args": ["evidence"],
            "optional_args": ["prior", "weights"],
```

```
        "accepts_kwargs": True,
        "description": "Calculate Bayesian confidence score",
    },
    "_extract_indicators": {
        "required_args": ["content"],
        "optional_args": ["indicator_patterns", "extraction_mode"],
        "accepts_kwargs": True,
        "description": "Extract KPI indicators from content",
    },
    "_parse_temporal_reference": {
        "required_args": ["text"],
        "optional_args": ["reference_date", "format_hints"],
        "accepts_kwargs": True,
        "description": "Parse temporal reference from text",
    },
    "_determine_policy_area": {
        "required_args": ["content"],
        "optional_args": ["taxonomy", "multi_label"],
        "accepts_kwargs": True,
        "description": "Classify content into policy area",
    },
    "_compile_regex_patterns": {
        "required_args": ["pattern_list"],
        "optional_args": ["flags", "validate"],
        "accepts_kwargs": False,
        "description": "Compile list of regex patterns",
    },
    "_analyze_source_reliability": {
        "required_args": ["source"],
        "optional_args": ["source_patterns", "reliability_threshold"],
        "accepts_kwargs": True,
        "description": "Analyze reliability of information source",
    },
    "_validate_numerical_consistency": {
        "required_args": ["numbers"],
        "optional_args": ["tolerance", "consistency_rules"],
        "accepts_kwargs": True,
        "description": "Validate numerical consistency across values",
    },
    "_calculate_bayesian_update": {
        "required_args": ["prior", "likelihood", "evidence"],
        "optional_args": ["normalization"],
        "accepts_kwargs": True,
        "description": "Calculate Bayesian posterior update",
    },
    "_extract_entities": {
        "required_args": ["content"],
        "optional_args": ["entity_types", "confidence_threshold"],
        "accepts_kwargs": True,
        "description": "Extract named entities from content",
    },
    "_parse_citation": {
        "required_args": ["text"],
        "optional_args": ["citation_style", "strict_mode"],
```

```
        "accepts_kwargs": True,
        "description": "Parse citation from text",
    },
    "_determine_validation_type": {
        "required_args": ["validation_spec"],
        "optional_args": ["context"],
        "accepts_kwargs": True,
        "description": "Determine type of validation to apply",
    },
    "_compile_indicator_patterns": {
        "required_args": ["indicators"],
        "optional_args": ["category", "weights"],
        "accepts_kwargs": False,
        "description": "Compile indicator patterns for matching",
    },
    "_analyze_coherence_score": {
        "required_args": ["content"],
        "optional_args": ["coherence_patterns", "scoring_mode"],
        "accepts_kwargs": True,
        "description": "Analyze narrative coherence score",
    },
    "_validate_threshold_compliance": {
        "required_args": ["value", "thresholds"],
        "optional_args": ["strict_mode"],
        "accepts_kwargs": True,
        "description": "Validate value against thresholds",
    },
    "_calculate_evidence_weight": {
        "required_args": ["evidence"],
        "optional_args": ["weighting_scheme", "normalization"],
        "accepts_kwargs": True,
        "description": "Calculate evidence weight for scoring",
    },
    "_extract_temporal_markers": {
        "required_args": ["content"],
        "optional_args": ["temporal_patterns", "extraction_depth"],
        "accepts_kwargs": True,
        "description": "Extract temporal markers from content",
    },
    "_parse_budget_allocation": {
        "required_args": ["text"],
        "optional_args": ["currency", "fiscal_year"],
        "accepts_kwargs": True,
        "description": "Parse budget allocation from text",
    },
    "_determine_risk_level": {
        "required_args": ["indicators"],
        "optional_args": ["risk_thresholds", "aggregation_method"],
        "accepts_kwargs": True,
        "description": "Determine risk level from indicators",
    },
    "_compile_validation_rules": {
        "required_args": ["rules"],
        "optional_args": ["rule_format"],
```

```
            "accepts_kwargs": False,
            "description": "Compile validation rules for execution",
        },
        "_analyze_stakeholder_impact": {
            "required_args": ["stakeholders", "policy"],
            "optional_args": ["impact_dimensions", "time_horizon"],
            "accepts_kwargs": True,
            "description": "Analyze stakeholder impact of policy",
        },
        "_validate_governance_structure": {
            "required_args": ["structure"],
            "optional_args": ["governance_standards", "strict_mode"],
            "accepts_kwargs": True,
            "description": "Validate governance structure compliance",
        },
        "_calculate_alignment_score": {
            "required_args": ["policy_content", "reference_framework"],
            "optional_args": ["alignment_weights", "scoring_method"],
            "accepts_kwargs": True,
            "description": "Calculate alignment score with framework",
        },
        "_extract_constraint_declarations": {
            "required_args": ["content"],
            "optional_args": ["constraint_types", "extraction_mode"],
            "accepts_kwargs": True,
            "description": "Extract constraint declarations from content",
        },
        "_parse_implementation_timeline": {
            "required_args": ["text"],
            "optional_args": ["reference_date", "granularity"],
            "accepts_kwargs": True,
            "description": "Parse implementation timeline from text",
        },
    }

    return routes

def route(
    self,
    class_name: str,
    method_name: str,
    payload: MutableMapping[str, Any],
) -> tuple[tuple[Any, ...], dict[str, Any]]:
    """
    Route method call with special handling and strict validation.

    This override:
    1. Checks for special route definitions
    2. Applies strict validation
    3. Prevents silent parameter drops
    4. Tracks metrics

    Args:
        class_name: Target class name
```

```
        method_name: Target method name
        payload: Method parameters

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

    Raises:
        ArgumentValidationError: On validation failure
    """
    with self._metrics_lock:
        self._metrics.total_routes += 1

    # Check for special route
    if method_name in self._special_routes:
        return self._route_special(class_name, method_name, payload)

    # Use default routing with enhanced validation
    return self._route_default_strict(class_name, method_name, payload)

def _route_special(
    self,
    class_name: str,
    method_name: str,
    payload: MutableMapping[str, Any],
) -> tuple[tuple[Any, ...], dict[str, Any]]:
    """
    Route using special route definition.

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

    Returns:
        Tuple of (args, kwargs)
    """
    with self._metrics_lock:
        self._metrics.special_routes_hit += 1

    route_spec = self._special_routes[method_name]
    required_args = set(route_spec["required_args"])
    optional_args = set(route_spec["optional_args"])
    accepts_kwargs = route_spec["accepts_kwargs"]

    provided_keys = set(payload.keys())

    # Check required arguments
    missing = required_args - provided_keys
    if missing:
        with self._metrics_lock:
            self._metrics.validation_errors += 1
        logger.error(
            "special_route_missing_args",
            class_name=class_name,
```

```
            method=method_name,
            missing=sorted(missing),
        )
        raise ArgumentValidationError(
            class_name,
            method_name,
            missing=missing,
        )

    # Check unexpected arguments
    expected = required_args | optional_args
    unexpected = provided_keys - expected

    if unexpected and not accepts_kwargs:
        # Method doesn't accept **kwargs, so unexpected args are an error
        with self._metrics_lock:
            self._metrics.validation_errors += 1
            self._metrics.silent_drops_prevented += 1

        logger.error(
            "special_route_unexpected_args",
            class_name=class_name,
            method=method_name,
            unexpected=sorted(unexpected),
            accepts_kwargs=accepts_kwargs,
        )
        raise ArgumentValidationError(
            class_name,
            method_name,
            unexpected=unexpected,
        )

    # Build kwargs (all parameters go to kwargs for special routes)
    kwargs = dict(payload)

    logger.debug(
        "special_route_applied",
        class_name=class_name,
        method=method_name,
        params_count=len(kwargs),
    )

    return (), kwargs

def _route_default_strict(
    self,
    class_name: str,
    method_name: str,
    payload: MutableMapping[str, Any],
) -> tuple[tuple[Any, ...], dict[str, Any]]:
    """
    Route using default strategy with strict validation.

    This prevents silent parameter drops by failing when:
```

```
- Required arguments are missing
- Unexpected arguments are provided AND method lacks **kwargs

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

Returns:
    Tuple of (args, kwargs)
"""
with self._metrics_lock:
    self._metrics.default_routes_hit += 1

# Use base implementation for inspection
spec = self.describe(class_name, method_name)

# Strict validation: if unexpected args and no **kwargs, fail
provided_keys = set(payload.keys())
accepted = set(spec.accepted_arguments)
unexpected = provided_keys - accepted

if unexpected and not spec.has_var_keyword:
    # Method doesn't accept **kwargs - unexpected args are errors
    with self._metrics_lock:
        self._metrics.validation_errors += 1
        self._metrics.silent_drops_prevented += 1

    logger.error(
        "default_route_unexpected_args_strict",
        class_name=class_name,
        method=method_name,
        unexpected=sorted(unexpected),
        has_var_keyword=spec.has_var_keyword,
    )
    raise ArgumentValidationError(
        class_name,
        method_name,
        unexpected=unexpected,
    )

# Delegate to base implementation
try:
    result = super().route(class_name, method_name, payload)
    logger.debug(
        "default_route_applied",
        class_name=class_name,
        method=method_name,
    )
    return result
except ArgumentValidationError:
    with self._metrics_lock:
        self._metrics.validation_errors += 1
    raise
```

```
def get_special_route_coverage(self) -> int:
    """
    Get count of special routes defined.

    Returns:
        Number of special routes (target: ?25)
    """
    return len(self._special_routes)

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

    Returns:
        Dict with routing statistics
    """
    total = self._metrics.total_routes or 1  # Avoid division by zero

    return {
        "total_routes": self._metrics.total_routes,
        "special_routes_hit": self._metrics.special_routes_hit,
        "special_routes_coverage": len(self._special_routes),
        "default_routes_hit": self._metrics.default_routes_hit,
        "validation_errors": self._metrics.validation_errors,
        "silent_drops_prevented": self._metrics.silent_drops_prevented,
        "special_route_hit_rate": self._metrics.special_routes_hit / total,
        "error_rate": self._metrics.validation_errors / total,
    }

def list_special_routes(self) -> list[dict[str, Any]]:
    """
    List all special routes with their specifications.

    Returns:
        List of route specifications
    """
    routes = []
    for method_name, spec in sorted(self._special_routes.items()):
        routes.append({
            "method_name": method_name,
            "required_args": spec["required_args"],
            "optional_args": spec["optional_args"],
            "accepts_kwargs": spec["accepts_kwargs"],
            "description": spec["description"],
        })
    return routes
```

src/farfan_pipeline/phases/Phase_two/base_executor_with_contract.py

```
from __future__ import annotations

import json
from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, Any

try:
    from jsonschema import Draft7Validator  # type: ignore
except Exception:  # pragma: no cover
    Draft7Validator = Any  # type: ignore[misc,assignment]

from canonic_phases.Phase_zero.paths import PROJECT_ROOT
# NEW: Replace legacy evidence modules with EvidenceNexus and Carver
from canonic_phases.Phase_two.evidence_nexus import EvidenceNexus, process_evidence
from canonic_phases.Phase_two.carver import DoctoralCarverSynthesizer

if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor
    from farfan_pipeline.core.types import PreprocessedDocument
else:  # pragma: no cover - runtime avoids import to break cycles
    MethodExecutor = Any
    PreprocessedDocument = Any


class BaseExecutorWithContract(ABC):
    """Contract-driven executor that routes all calls through MethodExecutor.

    Supports both v2 and v3 contract formats:
      - v2: Legacy format with method_inputs, assembly_rules, validation_rules at top
level
    - v3: New format with identity, executor_binding, method_binding, question_context,
          evidence_assembly, output_contract, validation_rules, etc.

      Contract version is auto-detected based on file name (.v3.json vs .json) and
structure.
    """

    _contract_cache: dict[str, dict[str, Any]] = {}
    _schema_validators: dict[str, Draft7Validator] = {}
    _factory_contracts_verified: bool = False
    _factory_verification_errors: list[str] = []

    def __init__(
        self,
        method_executor: MethodExecutor,
        signal_registry: Any,
        config: Any,
        questionnaire_provider: Any,
        calibration_orchestrator: Any | None = None,
        enriched_packs: dict[str, Any] | None = None,
        validation_orchestrator: Any | None = None,
    ) -> None:
```

```
    self.method_executor = method_executor
    self.signal_registry = signal_registry
    self.config = config
    self.questionnaire_provider = questionnaire_provider
    self.calibration_orchestrator = calibration_orchestrator
    # JOBFRONT 3: Support for enriched signal packs (intelligence layer)
    self.enriched_packs = enriched_packs or {}
    self._use_enriched_signals = len(self.enriched_packs) > 0
    # VALIDATION ORCHESTRATOR: Comprehensive validation tracking
    self.validation_orchestrator = validation_orchestrator
    self._use_validation_orchestrator = validation_orchestrator is not None

@classmethod
@abstractmethod
def get_base_slot(cls) -> str:
    raise NotImplementedError

@classmethod
def verify_all_base_contracts(
    cls, class_registry: dict[str, type[object]] | None = None
) -> dict[str, Any]:
    """Verify all 30 base executor contracts at factory initialization time.

    This method loads and validates all contracts for D1-Q1 through D6-Q5, checking:
    - Contract files exist and are valid JSON
    - Required fields are present (method_inputs/method_binding, assembly_rules,
      validation_rules, expected_elements)
    - JSON schema compliance (v2 or v3)
    - All referenced method classes exist in the class registry

    Args:
        class_registry: Optional class registry to verify method class existence.
                        If None, will attempt to import and build one.

    Returns:
        dict with keys:
            - passed: bool indicating if all contracts are valid
            - total_contracts: int count of contracts checked
            - errors: list of error messages for failed contracts
            - warnings: list of warning messages
            - verified_contracts: list of base_slot identifiers that passed

    Raises:
        RuntimeError: If verification fails with strict=True
    """
    if cls._factory_contracts_verified:
        return {
            "passed": len(cls._factory_verification_errors) == 0,
            "total_contracts": 30,
            "errors": cls._factory_verification_errors,
            "warnings": [],
            "verified_contracts": list(cls._contract_cache.keys()),
        }
```

```
        base_slots = [
            f"D{d}-Q{q}" for d in range(1, 7) for q in range(1, 6)
        ]

        if class_registry is None:
            try:
                from orchestration.class_registry import (
                    build_class_registry,
                )
                class_registry = build_class_registry()
            except Exception as exc:
                cls._factory_verification_errors.append(
                    f"Failed to build class registry for verification: {exc}"
                )

        errors: list[str] = []
        warnings: list[str] = []
        verified_contracts: list[str] = []

        for base_slot in base_slots:
            try:
                result = cls._verify_single_contract(base_slot, class_registry)
                if result["passed"]:
                    verified_contracts.append(base_slot)
                else:
                    errors.extend(
                        f"[{base_slot}] {err}" for err in result["errors"]
                    )
                warnings.extend(
                    f"[{base_slot}] {warn}" for warn in result.get("warnings", [])
                )
            except Exception as exc:
                     errors.append(f"[{base_slot}] Unexpected error during verification:
{exc}")

        cls._factory_contracts_verified = True
        cls._factory_verification_errors = errors

        return {
            "passed": len(errors) == 0,
            "total_contracts": len(base_slots),
            "errors": errors,
            "warnings": warnings,
            "verified_contracts": verified_contracts,
        }

    @classmethod
    def _verify_single_contract(
        cls, base_slot: str, class_registry: dict[str, type[object]] | None = None
    ) -> dict[str, Any]:
        """Verify a single contract for completeness and validity.

        Args:
            base_slot: Base slot identifier (e.g., "D1-Q1")
```

```
            class_registry: Optional class registry for method class verification

        Returns:
            dict with keys:
                - passed: bool
                - errors: list of error messages
                - warnings: list of warning messages
                - contract_version: detected version (v2/v3)
                - contract_path: path to contract file
        """
        errors: list[str] = []
        warnings: list[str] = []

        dimension = int(base_slot[1])
        question = int(base_slot[4])
        q_number = (dimension - 1) * 5 + question
        q_id = f"Q{q_number:03d}"

              contracts_dir = PROJECT_ROOT / "src" / "farfan_pipeline" / "phases" /
"Phase_two" / "json_files_phase_two" / "executor_contracts"

        v3_path = contracts_dir / f"{base_slot}.v3.json"
        v2_path = contracts_dir / f"{base_slot}.json"
        v3_specialized_path = contracts_dir / "specialized" / f"{q_id}.v3.json"
        v2_specialized_path = contracts_dir / "specialized" / f"{q_id}.json"

        contract_path = None
        if v3_path.exists():
            contract_path = v3_path
            expected_version = "v3"
        elif v2_path.exists():
            contract_path = v2_path
            expected_version = "v2"
        elif v3_specialized_path.exists():
            contract_path = v3_specialized_path
            expected_version = "v3"
        elif v2_specialized_path.exists():
            contract_path = v2_specialized_path
            expected_version = "v2"
        else:
            errors.append(
                                f"Contract file not found. Tried: {v3_path}, {v2_path},
{v3_specialized_path}, {v2_specialized_path}"
            )
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": None,
                "contract_path": None,
            }

        try:
            contract = json.loads(contract_path.read_text(encoding="utf-8"))
```

```
        except json.JSONDecodeError as exc:
            errors.append(f"Invalid JSON in contract file: {exc}")
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": expected_version,
                "contract_path": str(contract_path),
            }
        except Exception as exc:
            errors.append(f"Failed to read contract file: {exc}")
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": expected_version,
                "contract_path": str(contract_path),
            }

        detected_version = cls._detect_contract_version(contract)
        if detected_version != expected_version:
            warnings.append(
                    f"Contract structure is {detected_version} but file naming suggests
{expected_version}"
            )

        try:
            validator = cls._get_schema_validator(detected_version)
                  schema_errors = sorted(validator.iter_errors(contract), key=lambda e:
e.path)
            if schema_errors:
                errors.extend(
                      f"Schema validation error: {err.message} at {'.'.join(str(p) for p
in err.path)}"
                    for err in schema_errors[:10]
                )
        except FileNotFoundError as exc:
                    warnings.append(f"Schema file not found: {exc}. Skipping schema
validation.")
        except Exception as exc:
            warnings.append(f"Schema validation error: {exc}")

        if detected_version == "v3":
                      v3_errors = cls._verify_v3_contract_fields(contract, base_slot,
class_registry)
            errors.extend(v3_errors)
        else:
                      v2_errors = cls._verify_v2_contract_fields(contract, base_slot,
class_registry)
            errors.extend(v2_errors)

        return {
            "passed": len(errors) == 0,
            "errors": errors,
```

```
            "warnings": warnings,
            "contract_version": detected_version,
            "contract_path": str(contract_path),
        }

    @classmethod
    def _verify_v2_contract_fields(
        cls,
        contract: dict[str, Any],
        base_slot: str,
        class_registry: dict[str, type[object]] | None = None,
    ) -> list[str]:
        """Verify required fields for v2 contract format.

        Args:
            contract: Parsed contract dict
            base_slot: Base slot identifier
            class_registry: Optional class registry for method verification

        Returns:
            List of error messages (empty if all checks pass)
        """
        errors: list[str] = []

        if "method_inputs" not in contract:
            errors.append("Missing required field: method_inputs")
        elif not isinstance(contract["method_inputs"], list):
            errors.append("method_inputs must be a list")
        else:
            method_inputs = contract["method_inputs"]
            if not method_inputs:
                errors.append("method_inputs is empty")
            else:
                for idx, method_spec in enumerate(method_inputs):
                    if not isinstance(method_spec, dict):
                        errors.append(f"method_inputs[{idx}] is not a dict")
                        continue
                    if "class" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'class' field")
                    if "method" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'method' field")

                    if class_registry is not None and "class" in method_spec:
                        class_name = method_spec["class"]
                        if class_name not in class_registry:
                            errors.append(
                                f"method_inputs[{idx}]: class '{class_name}' not found
in class registry"
                            )

        if "assembly_rules" not in contract:
            errors.append("Missing required field: assembly_rules")
        elif not isinstance(contract["assembly_rules"], list):
            errors.append("assembly_rules must be a list")
```

```
        if "validation_rules" not in contract:
            errors.append("Missing required field: validation_rules")

        return errors

    @classmethod
    def _verify_v3_contract_fields(
        cls,
        contract: dict[str, Any],
        base_slot: str,
        class_registry: dict[str, type[object]] | None = None,
    ) -> list[str]:
        """Verify required fields for v3 contract format.

        Args:
            contract: Parsed contract dict
            base_slot: Base slot identifier
            class_registry: Optional class registry for method verification

        Returns:
            List of error messages (empty if all checks pass)
        """
        errors: list[str] = []

        if "identity" not in contract:
            errors.append("Missing required field: identity")
        else:
            identity = contract["identity"]
            if "base_slot" not in identity:
                errors.append("identity missing 'base_slot' field")
            elif identity["base_slot"] != base_slot:
                errors.append(
                                f"identity.base_slot mismatch: expected {base_slot}, got
{identity['base_slot']}"
                )

        if "method_binding" not in contract:
            errors.append("Missing required field: method_binding")
        else:
            method_binding = contract["method_binding"]
                        orchestration_mode = method_binding.get("orchestration_mode",
"single_method")

            if orchestration_mode == "multi_method_pipeline":
                if "methods" not in method_binding:
                                errors.append("method_binding missing 'methods' array for
multi_method_pipeline mode")
                elif not isinstance(method_binding["methods"], list):
                    errors.append("method_binding.methods must be a list")
                else:
                    methods = method_binding["methods"]
                    if not methods:
                        errors.append("method_binding.methods is empty")
```

```
                    else:
                        for idx, method_spec in enumerate(methods):
                            if not isinstance(method_spec, dict):
                                errors.append(f"methods[{idx}] is not a dict")
                                continue
                            if "class_name" not in method_spec:
                                    errors.append(f"methods[{idx}] missing 'class_name'
field")
                            if "method_name" not in method_spec:
                                    errors.append(f"methods[{idx}] missing 'method_name'
field")

                                    if class_registry is not None and "class_name" in
method_spec:
                                class_name = method_spec["class_name"]
                                if class_name not in class_registry:
                                    errors.append(
                                        f"methods[{idx}]: class '{class_name}' not found
in class registry"
                                    )
                      elif "class_name" not in method_binding and "primary_method" not in
method_binding:
                errors.append(
                            "method_binding missing 'class_name' or 'primary_method' for
single_method mode"
                )
            else:
                class_name = method_binding.get("class_name")
                if not class_name and "primary_method" in method_binding:
                    class_name = method_binding["primary_method"].get("class_name")

                if class_name and class_registry is not None:
                    if class_name not in class_registry:
                        errors.append(
                               f"method_binding: class '{class_name}' not found in class
registry"
                        )

        if "evidence_assembly" not in contract:
            errors.append("Missing required field: evidence_assembly")
        else:
            evidence_assembly = contract["evidence_assembly"]
            if "assembly_rules" not in evidence_assembly:
                errors.append("evidence_assembly missing 'assembly_rules' field")
            elif not isinstance(evidence_assembly["assembly_rules"], list):
                errors.append("evidence_assembly.assembly_rules must be a list")

        if "validation_rules" not in contract:
            errors.append("Missing required field: validation_rules")

        if "question_context" not in contract:
            errors.append("Missing required field: question_context")
        else:
            question_context = contract["question_context"]
```

```
            if "expected_elements" not in question_context:
                errors.append("question_context missing 'expected_elements' field")

        if "error_handling" not in contract:
            errors.append("Missing required field: error_handling")

        return errors

    @classmethod
    def _get_schema_validator(cls, version: str = "v2") -> Draft7Validator:
        """Get schema validator for the specified contract version.

        Args:
            version: Contract version ("v2" or "v3")

        Returns:
            Draft7Validator for the specified version
        """
        if version not in cls._schema_validators:
            # Fallback for schema path (user reported misconfiguration)
            if version == "v3":
                schema_path = (
                    PROJECT_ROOT
                    / "config"
                    / "schemas"
                    / "executor_contract.v3.schema.json"
                )
            else:
                schema_path = PROJECT_ROOT / "config" / "executor_contract.schema.json"

                            #  If  default  path  doesn't  exist,  try  local  path  in
Phase_two/json_files_phase_two
            if not schema_path.exists():
                local_path = (
                    PROJECT_ROOT
                    / "src"
                    / "canonic_phases"
                    / "Phase_two"
                    / "json_files_phase_two"
                    / f"executor_contract.{version}.schema.json"
                )
                if local_path.exists():
                    schema_path = local_path
                else:
                     # Attempt to construct minimal schema in memory if files missing
                     # to prevent crashing if schema assets are misplaced
                     import logging
                             logging.warning(f"Schema file missing at {schema_path} and
{local_path}. Using minimal fallback.")
                     minimal_schema = {"type": "object", "additionalProperties": True}
                     cls._schema_validators[version] = Draft7Validator(minimal_schema)
                     return cls._schema_validators[version]

            if not schema_path.exists():
```

```
                raise FileNotFoundError(f"Contract schema not found: {schema_path}")
            schema = json.loads(schema_path.read_text(encoding="utf-8"))
            cls._schema_validators[version] = Draft7Validator(schema)
        return cls._schema_validators[version]

    @classmethod
    def _detect_contract_version(cls, contract: dict[str, Any]) -> str:
        """Detect contract version from structure.

        v3 contracts have: identity, executor_binding, method_binding, question_context
        v2 contracts have: method_inputs, assembly_rules at top level

        Returns:
            "v3" or "v2"
        """
        v3_indicators = [
            "identity",
            "executor_binding",
            "method_binding",
            "question_context",
        ]
        if all(key in contract for key in v3_indicators):
            return "v3"
        return "v2"

    @classmethod
    def _load_contract(cls, question_id: str | None = None) -> dict[str, Any]:
        base_slot = cls.get_base_slot()

            # Use specific question_id if provided, otherwise derive base Q-id from
base_slot
        if question_id:
            cache_key = f"{base_slot}:{question_id}"
            q_id = question_id
        else:
            cache_key = base_slot
            dimension = int(base_slot[1])
            question = int(base_slot[4])
            q_number = (dimension - 1) * 5 + question
            q_id = f"Q{q_number:03d}"

        if cache_key in cls._contract_cache:
            return cls._contract_cache[cache_key]

              contracts_dir = PROJECT_ROOT / "src" / "farfan_pipeline" / "phases" /
"Phase_two" / "json_files_phase_two" / "executor_contracts"

        v3_path = contracts_dir / f"{base_slot}.v3.json"
        v2_path = contracts_dir / f"{base_slot}.json"
        v3_specialized_path = contracts_dir / "specialized" / f"{q_id}.v3.json"
        v2_specialized_path = contracts_dir / "specialized" / f"{q_id}.json"

        if v3_specialized_path.exists():
            contract_path = v3_specialized_path
```

```
            expected_version = "v3"
        elif v2_specialized_path.exists():
            contract_path = v2_specialized_path
            expected_version = "v2"
        elif v3_path.exists():
            contract_path = v3_path
            expected_version = "v3"
        elif v2_path.exists():
            contract_path = v2_path
            expected_version = "v2"
        else:
            raise FileNotFoundError(
                f"Contract not found for {base_slot} / {q_id}. "
                                f"Tried:  {v3_path},  {v2_path},  {v3_specialized_path},
{v2_specialized_path}"
            )

        contract = json.loads(contract_path.read_text(encoding="utf-8"))

        # Detect actual version from structure
        detected_version = cls._detect_contract_version(contract)
        if detected_version != expected_version:
            import logging

            logging.warning(
                f"Contract {contract_path.name} has structure of {detected_version} "
                f"but file naming suggests {expected_version}"
            )

        # Validate with appropriate schema
        validator = cls._get_schema_validator(detected_version)
        errors = sorted(validator.iter_errors(contract), key=lambda e: e.path)
        if errors:
            messages = "; ".join(err.message for err in errors)
            raise ValueError(
                    f"Contract  validation  failed  for  {base_slot}  ({detected_version}):
{messages}"
            )

        # Tag contract with version for later use
        contract["_contract_version"] = detected_version

        contract_version = contract.get("contract_version")
        if contract_version and not str(contract_version).startswith("2"):
            raise ValueError(
                    f"Unsupported  contract_version  {contract_version}  for  {base_slot};
expected v2.x"
            )

        identity_base_slot = contract.get("identity", {}).get("base_slot")
        if identity_base_slot and identity_base_slot != base_slot:
            raise ValueError(
                        f"Contract  base_slot  mismatch:  expected  {base_slot},  found
{identity_base_slot}"
```

```
            )

        cls._contract_cache[cache_key] = contract
        return contract

    def _validate_signal_requirements(
        self,
        signal_pack: Any,
        signal_requirements: dict[str, Any],
        base_slot: str,
    ) -> None:
        """Validate that signal requirements from contract are met.

        Args:
            signal_pack: Signal pack retrieved from registry (may be None)
            signal_requirements: signal_requirements section from contract
            base_slot: Base slot identifier for error messages

        Raises:
            RuntimeError: If mandatory signal requirements are not met
        """
        mandatory_signals = signal_requirements.get("mandatory_signals", [])
        minimum_threshold = signal_requirements.get("minimum_signal_threshold", 0.0)

        # Check if mandatory signals are required but no signal pack available
        if mandatory_signals and signal_pack is None:
            raise RuntimeError(
                f"Contract {base_slot} requires mandatory signals {mandatory_signals}, "
                "but no signal pack was retrieved from registry. "
                   "Ensure signal registry is properly configured and policy_area_id is
valid."
            )

        # If signal pack exists, validate signal strength
        if signal_pack is not None and minimum_threshold > 0:
            # Check if signal pack has strength attribute
            if hasattr(signal_pack, "strength") or (
                isinstance(signal_pack, dict) and "strength" in signal_pack
            ):
                strength = (
                    signal_pack.strength
                    if hasattr(signal_pack, "strength")
                    else signal_pack["strength"]
                )
                if strength < minimum_threshold:
                    raise RuntimeError(
                                f"Contract {base_slot} requires minimum signal threshold
{minimum_threshold}, "
                        f"but signal pack has strength {strength}. "
                        "Signal quality is insufficient for execution."
                    )

    @staticmethod
    def _set_nested_value(
```

```
        target_dict: dict[str, Any], key_path: str, value: Any
    ) -> None:
        """Set a value in a nested dict using dot-notation key path.

        Args:
            target_dict: The dictionary to modify
            key_path: Dot-separated path (e.g., "text_mining.critical_links")
            value: The value to set

        Example:
            _set_nested_value(d, "a.b.c", 123) ? d["a"]["b"]["c"] = 123
        """
        keys = key_path.split(".")
        current = target_dict

        # Navigate to the parent of the final key, creating dicts as needed
        for key in keys[:-1]:
            if key not in current:
                current[key] = {}
            elif not isinstance(current[key], dict):
                # Key exists but is not a dict, cannot nest further
                raise ValueError(
                    f"Cannot set nested value at '{key_path}': "
                    f"intermediate key '{key}' exists but is not a dict"
                )
            current = current[key]

        # Set the final key
        current[keys[-1]] = value

    def _check_failure_contract(
        self, evidence: dict[str, Any], error_handling: dict[str, Any]
    ) -> None:
        failure_contract = error_handling.get("failure_contract", {})
        abort_conditions = failure_contract.get("abort_if", [])
        if not abort_conditions:
            return

        emit_code = failure_contract.get("emit_code", "GENERIC_ABORT")

        for condition in abort_conditions:
            # Example condition check. This could be made more sophisticated.
            if condition == "missing_required_element" and evidence.get(
                "validation", {}
            ).get("errors"):
                 # This logic assumes errors from the validator imply a missing required
element,
                # which is true with our new validator.
                raise ValueError(
                      f"Execution aborted by failure contract due to '{condition}'. Emit
code: {emit_code}"
                )
            if condition == "incomplete_text" and not evidence.get("metadata", {}).get(
                "text_complete", True
```

```
            ):
                raise ValueError(
                      f"Execution aborted by failure contract due to '{condition}'. Emit
code: {emit_code}"
                )

    def execute(
        self,
        document: PreprocessedDocument,
        method_executor: MethodExecutor,
        *,
        question_context: dict[str, Any],
    ) -> dict[str, Any]:
        if method_executor is not self.method_executor:
            raise RuntimeError(
                "Mismatched MethodExecutor instance for contract executor"
            )

        base_slot = self.get_base_slot()
        if question_context.get("base_slot") != base_slot:
            raise ValueError(
                 f"Question base_slot {question_context.get('base_slot')} does not match
executor {base_slot}"
            )

        question_id = question_context.get("question_id")
        contract = self._load_contract(question_id=question_id)
        contract_version = contract.get("_contract_version", "v2")

        if contract_version == "v3":
            return self._execute_v3(document, question_context, contract)
        else:
            return self._execute_v2(document, question_context, contract)

    def _execute_v2(
        self,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
        contract: dict[str, Any],
    ) -> dict[str, Any]:
        """Execute using v2 contract format (legacy)."""
        base_slot = self.get_base_slot()
        question_id = question_context.get("question_id")
        question_global = question_context.get("question_global")
        policy_area_id = question_context.get("policy_area_id")
        identity = question_context.get("identity", {})
        patterns = question_context.get("patterns", [])
        expected_elements = question_context.get("expected_elements", [])

        # JOBFRONT 3: Use enriched signal packs if available
        signal_pack = None
        enriched_pack = None
        applicable_patterns = patterns  # Default to contract patterns
        document_context = {}
```

```
        if self._use_enriched_signals and policy_area_id in self.enriched_packs:
            # Use enriched intelligence layer
            enriched_pack = self.enriched_packs[policy_area_id]
            signal_pack = enriched_pack.base_pack  # Maintain compatibility

            # Create document context from available metadata
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_intelligence_layer
import (
                create_document_context,
            )

            doc_metadata = getattr(document, "metadata", {})
            document_context = create_document_context(
                section=doc_metadata.get("section"),
                chapter=doc_metadata.get("chapter"),
                page=doc_metadata.get("page"),
                policy_area=policy_area_id
            )

            # Get context-filtered patterns (REFACTORING #6: context scoping)
                                                        applicable_patterns     =
enriched_pack.get_patterns_for_context(document_context)

            # Expand patterns semantically (REFACTORING #2: semantic expansion)
            if applicable_patterns and isinstance(applicable_patterns[0], dict):
                 pattern_strings = [p.get('pattern', p) if isinstance(p, dict) else p for
p in applicable_patterns]
            else:
                pattern_strings = applicable_patterns

            expanded_patterns = enriched_pack.expand_patterns(pattern_strings)
            applicable_patterns = expanded_patterns

          elif self.signal_registry is not None and hasattr(self.signal_registry, "get")
and policy_area_id:
            # Fallback to legacy signal registry
            signal_pack = self.signal_registry.get(policy_area_id)

        common_kwargs: dict[str, Any] = {
            "document": document,
            "base_slot": base_slot,
            "raw_text": getattr(document, "raw_text", None),
            "text": getattr(document, "raw_text", None),
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": identity.get("dimension_id"),
            "cluster_id": identity.get("cluster_id"),
            "signal_pack": signal_pack,
            "enriched_pack": enriched_pack,  # NEW: Pass enriched pack
            "document_context": document_context,  # NEW: Pass document context
            "question_patterns": applicable_patterns,  # Use filtered/expanded patterns
```

```
            "expected_elements": expected_elements,
        }

        method_outputs: dict[str, Any] = {}
        method_inputs = contract.get("method_inputs", [])
        indexed = list(enumerate(method_inputs))
        sorted_inputs = sorted(
            indexed, key=lambda pair: (pair[1].get("priority", 2), pair[0])
        )

        calibration_results = {}

        for _, entry in sorted_inputs:
            class_name = entry["class"]
            method_name = entry["method"]
            provides = entry.get("provides", [])
            extra_args = entry.get("args", {})

            payload = {**common_kwargs, **extra_args}

            method_id = f"{class_name}.{method_name}"

            if self.calibration_orchestrator:
                try:
                                                                                from
cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator
import (
                        MethodBelowThresholdError,
                    )

                    calibration_result = self.calibration_orchestrator.calibrate(
                        method_id=method_id,
                        context=payload,
                        evidence=None
                    )

                    calibration_results[method_id] = calibration_result.to_dict()

                    import logging
                    logger = logging.getLogger(__name__)
                    logger.info(
                                            f"[{base_slot}]  Calibration:  {method_id}  ?
{calibration_result.final_score:.3f}"
                    )

                except MethodBelowThresholdError as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    logger.error(
                        f"[{base_slot}] Method {method_id} FAILED calibration: "
                        f"score={e.score:.3f}, threshold={e.threshold:.3f}"
                    )
                    raise RuntimeError(
                        f"Method {method_id} failed calibration threshold"
```

```
                    ) from e
                except Exception as e:
                    import logging
                    logger = logging.getLogger(__name__)
                       logger.warning(f"[{base_slot}] Calibration error for {method_id}:
{e}")

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

            if "signal_pack" in payload and payload["signal_pack"] is not None:
                if "_signal_usage" not in method_outputs:
                    method_outputs["_signal_usage"] = []
                method_outputs["_signal_usage"].append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": payload["signal_pack"].policy_area,
                        "version": payload["signal_pack"].version,
                    }
                )

            if isinstance(provides, str):
                method_outputs[provides] = result
            else:
                for key in provides:
                    method_outputs[key] = result

        assembly_rules = contract.get("assembly_rules", [])

        # NEW: Use EvidenceNexus instead of legacy EvidenceAssembler
        nexus_result = process_evidence(
            method_outputs=method_outputs,
            assembly_rules=assembly_rules,
            validation_rules=contract.get("validation_rules", []),
            question_context={
                "question_id": question_id,
                "question_global": question_global,
                "expected_elements": expected_elements,
                "patterns": applicable_patterns,
                       # Provide raw text so EvidenceNexus can run pattern extraction
deterministically.
                "raw_text": getattr(document, "raw_text", "") or "",
            },
            signal_pack=signal_pack,  # SISAS: Enable signal provenance
            contract=contract,
        )

        evidence = nexus_result["evidence"]
        trace = nexus_result["trace"]
        validation = nexus_result["validation"]
```

```
        # JOBFRONT 3: Extract structured evidence if enriched pack available
        completeness = 1.0
        missing_elements = []
        patterns_used = []

        if enriched_pack is not None and expected_elements:
            # Build signal node for evidence extraction
            signal_node = {
                "id": question_id,
                "expected_elements": expected_elements,
                "patterns": applicable_patterns,
                "validations": contract.get("validation_rules", [])
            }

            # Extract structured evidence (REFACTORING #5: evidence structure)
            evidence_result = enriched_pack.extract_evidence(
                text=getattr(document, "raw_text", ""),
                signal_node=signal_node,
                document_context=document_context
            )

            # Merge structured evidence into result
            for element_type, matches in evidence_result.evidence.items():
                if element_type not in evidence:
                    evidence[element_type] = matches

            completeness = evidence_result.completeness
            missing_elements = evidence_result.missing_required

            # Track patterns used (for confidence calculation)
            if isinstance(applicable_patterns, list):
                patterns_used = [p.get('id', p) if isinstance(p, dict) else p
                                 for p in applicable_patterns[:10]]  # Top 10

        # Note: Validation is now handled by EvidenceNexus above
        error_handling = contract.get("error_handling", {})

        # JOBFRONT 3: Add contract validation if enriched pack available
        contract_validation = None
        if enriched_pack is not None:
            # Build signal node for contract validation
            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules,
                "expected_elements": expected_elements
            }

            # Validate with contracts (REFACTORING #4: contract validation)
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator
import (
                validate_result_with_orchestrator,
            )
```

```
            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                                            orchestrator=self.validation_orchestrator   if
self._use_validation_orchestrator else None,
                auto_register=self._use_validation_orchestrator
            )

            # Merge contract validation into standard validation
            if not contract_validation.passed:
                validation["status"] = "failed"
                validation["errors"] = validation.get("errors", [])
                validation["errors"].append({
                    "error_code": contract_validation.error_code,
                    "condition_violated": contract_validation.condition_violated,
                    "remediation": contract_validation.remediation,
                    "failures_detailed": [
                        {
                            "type": f.failure_type,
                            "field": f.field_name,
                            "message": f.message,
                            "severity": f.severity,
                            "remediation": f.remediation
                        }
                        for f in contract_validation.failures_detailed[:5]
                    ]
                })
                validation["contract_failed"] = True
                validation["contract_validation_details"] = {
                    "error_code": contract_validation.error_code,
                    "diagnostics": contract_validation.diagnostics,
                    "total_failures": len(contract_validation.failures_detailed)
                }
        elif self._use_validation_orchestrator:
                 # Even without enriched pack, use validation orchestrator with basic
validation
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator
import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": {"rules": validation_rules},
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator,
```

```
                auto_register=True
            )
        if error_handling:
            evidence_with_validation = {**evidence, "validation": validation}
            self._check_failure_contract(evidence_with_validation, error_handling)

        human_answer_template = contract.get("human_answer_template", "")
        human_answer = ""
        if human_answer_template:
            try:
                human_answer = human_answer_template.format(**evidence)
            except KeyError as e:
                      human_answer = f"Error formatting human answer: Missing key {e}.
Template: '{human_answer_template}'"
                import logging

                logging.warning(human_answer)

        result = {
            "base_slot": base_slot,
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": identity.get("dimension_id"),
            "cluster_id": identity.get("cluster_id"),
            "evidence": evidence,
            "validation": validation,
            "trace": trace,
            "human_answer": human_answer,
            # JOBFRONT 3: Add intelligence layer metadata
            "completeness": completeness,
            "missing_elements": missing_elements,
            "patterns_used": patterns_used,
            "enriched_signals_enabled": enriched_pack is not None,
            # VALIDATION ORCHESTRATOR: Add validation tracking metadata
            "contract_validation": {
                "enabled": contract_validation is not None,
                "passed": contract_validation.passed if contract_validation else None,
                 "error_code": contract_validation.error_code if contract_validation else
None,
                          "failure_count":  len(contract_validation.failures_detailed)  if
contract_validation else 0,
                "orchestrator_registered": self._use_validation_orchestrator
            },
            # CALIBRATION: Add calibration metadata
            "calibration_metadata": {
                "enabled": self.calibration_orchestrator is not None,
                "results": calibration_results,
                "summary": {
                    "total_methods": len(calibration_results),
                    "average_score": sum(
                        cr["final_score"] for cr in calibration_results.values()
                    ) / len(calibration_results) if calibration_results else 0.0,
                    "min_score": min(
```

```
                    (cr["final_score"] for cr in calibration_results.values()),
                    default=0.0
                ),
                "max_score": max(
                    (cr["final_score"] for cr in calibration_results.values()),
                    default=0.0
                ),
            }
        }
    }

    return result

def _execute_v3(
    self,
    document: PreprocessedDocument,
    question_context_external: dict[str, Any],
    contract: dict[str, Any],
) -> dict[str, Any]:
    """Execute using v3 contract format.

    In v3, contract contains all context, so we use contract['question_context']
    instead of question_context_external (which comes from orchestrator).
    """
    # Extract identity from contract
    identity = contract["identity"]
    base_slot = identity["base_slot"]
    question_id = identity["question_id"]
    dimension_id = identity["dimension_id"]
    policy_area_id = identity["policy_area_id"]

    # CALIBRATION ENFORCEMENT: Verify calibration status before execution
    calibration = contract.get("calibration", {})
    calibration_status = calibration.get("status", "placeholder")
    if calibration_status == "placeholder":
        import logging
        logging.info(
            f"Contract {base_slot} has placeholder calibration. "
            "Injecting live calibration parameters from UnitOfAnalysisLoader..."
        )
        # Override status to enable execution with alive parameters
        calibration["status"] = "calibrated_alive"
                        calibration["note"]  =  "Live  parameters  injected  from
canonic_description_unit_analysis.json"

    # Extract question context from contract (source of truth for v3)
    question_context = contract["question_context"]
    question_global = question_context_external.get(
        "question_global"
    )  # May come from orchestrator
    patterns = question_context.get("patterns", [])
    expected_elements = question_context.get("expected_elements", [])

    # Signal pack
```

```
        signal_pack = None
        if (
            self.signal_registry is not None
            and hasattr(self.signal_registry, "get")
            and policy_area_id
        ):
            signal_pack = self.signal_registry.get(policy_area_id)

        # SISAS: Inject consumption tracking (utility + proof chain)
        consumption_tracker = None
        try:
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_consumption_integrati
on import (
                inject_consumption_tracking,
            )

            consumption_tracker = inject_consumption_tracking(
                executor=self,
                question_id=question_id,
                policy_area_id=policy_area_id,
                # Deterministic: do not depend on wall clock time for proofs.
                injection_time=0.0,
            )
        except Exception:
            consumption_tracker = None

        # Build document context (for scope coherence + context-aware pattern filtering)
        document_context: dict[str, Any] = {}
        try:
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_context_scoper  import
(
                create_document_context,
            )

            doc_metadata = getattr(document, "metadata", {}) or {}
            document_context = create_document_context(
                section=doc_metadata.get("section"),
                chapter=doc_metadata.get("chapter"),
                page=doc_metadata.get("page"),
                policy_area=policy_area_id,
            )
        except Exception:
            document_context = {"policy_area": policy_area_id}

        # SIGNAL REQUIREMENTS VALIDATION: Verify signal requirements from contract
        signal_requirements = contract.get("signal_requirements", {})
        if signal_requirements:
            self._validate_signal_requirements(
                signal_pack, signal_requirements, base_slot
            )

        # Extract method binding
```

```
        method_binding = contract["method_binding"]
        orchestration_mode = method_binding.get("orchestration_mode", "single_method")

        # Prepare common kwargs
        common_kwargs: dict[str, Any] = {
            "document": document,
            "base_slot": base_slot,
            "raw_text": getattr(document, "raw_text", None),
            "text": getattr(document, "raw_text", None),
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": dimension_id,
            "cluster_id": identity.get("cluster_id"),
            "signal_pack": signal_pack,
            "question_patterns": patterns,
            "expected_elements": expected_elements,
            "question_context": question_context,
        }

        # Execute methods based on orchestration mode
        method_outputs: dict[str, Any] = {}
        signal_usage_list: list[dict[str, Any]] = []
        calibration_results: dict[str, Any] = {}

        if orchestration_mode == "multi_method_pipeline":
            # Multi-method execution: process all methods in priority order
            methods = method_binding.get("methods", [])
            if not methods:
                raise ValueError(
                    f"orchestration_mode is 'multi_method_pipeline' but no methods array
found in method_binding for {base_slot}"
                )

            # Sort by priority (lower priority number = execute first)
            sorted_methods = sorted(methods, key=lambda m: m.get("priority", 99))

            for method_spec in sorted_methods:
                class_name = method_spec["class_name"]
                method_name = method_spec["method_name"]
                provides = method_spec.get("provides", f"{class_name}.{method_name}")
                priority = method_spec.get("priority", 99)

                method_id = f"{class_name}.{method_name}"

                if self.calibration_orchestrator:
                    try:
                                                                        from
cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator
import (
                            MethodBelowThresholdError,
                        )

                        calibration_result = self.calibration_orchestrator.calibrate(
```

```
                            method_id=method_id,
                            context=common_kwargs,
                            evidence=None
                        )

                        calibration_results[method_id] = calibration_result.to_dict()

                        import logging
                        logger = logging.getLogger(__name__)
                        logger.info(
                                                f"[{base_slot}] Calibration: {method_id} ?
{calibration_result.final_score:.3f}"
                        )

                    except MethodBelowThresholdError as e:
                        import logging
                        logger = logging.getLogger(__name__)
                        logger.error(
                            f"[{base_slot}] Method {method_id} FAILED calibration: "
                            f"score={e.score:.3f}, threshold={e.threshold:.3f}"
                        )
                        raise RuntimeError(
                            f"Method {method_id} failed calibration threshold"
                        ) from e
                    except Exception as e:
                        import logging
                        logger = logging.getLogger(__name__)
                                    logger.warning(f"[{base_slot}] Calibration error for
{method_id}: {e}")

                try:
                    result = self.method_executor.execute(
                        class_name=class_name,
                        method_name=method_name,
                        **common_kwargs,
                    )

                                    # Store result using nested key structure (e.g.,
"text_mining.critical_links")
                    self._set_nested_value(method_outputs, provides, result)

                    # Track signal usage for this method
                    if signal_pack is not None:
                        signal_usage_list.append(
                            {
                                "method": f"{class_name}.{method_name}",
                                "policy_area": signal_pack.policy_area,
                                "version": signal_pack.version,
                                "priority": priority,
                            }
                        )

                except Exception as exc:
                    import logging
```

```
                    logging.error(
                                f"Method execution failed in multi-method pipeline:
{class_name}.{method_name}",
                        exc_info=True,
                    )
                    # Store error in trace for debugging
                    # Store error in a flat structure under _errors[provides]
                    if "_errors" not in method_outputs or not isinstance(
                        method_outputs["_errors"], dict
                    ):
                        method_outputs["_errors"] = {}
                    method_outputs["_errors"][provides] = {
                        "error": str(exc),
                        "method": f"{class_name}.{method_name}",
                    }
                    # Re-raise if error_handling policy requires it
                    error_handling = contract.get("error_handling", {})
                    on_method_failure = error_handling.get(
                        "on_method_failure", "propagate_with_trace"
                    )
                    if on_method_failure == "raise":
                        raise
                    # Otherwise continue with other methods

            else:
                # Single-method execution (backward compatible, default)
                class_name = method_binding.get("class_name")
                method_name = method_binding.get("method_name")

                if not class_name or not method_name:
                    # Try primary_method if direct class_name/method_name not found
                    primary_method = method_binding.get("primary_method", {})
                    class_name = primary_method.get("class_name") or class_name
                    method_name = primary_method.get("method_name") or method_name

                if not class_name or not method_name:
                    raise ValueError(
                            f"Invalid method_binding for {base_slot}: missing class_name or
method_name"
                    )

                method_id = f"{class_name}.{method_name}"

                if self.calibration_orchestrator:
                    try:
                                                                                from
cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator
import (
                            MethodBelowThresholdError,
                        )

                        calibration_result = self.calibration_orchestrator.calibrate(
                            method_id=method_id,
```

```
                        context=common_kwargs,
                        evidence=None
                    )

                    calibration_results[method_id] = calibration_result.to_dict()

                    import logging
                    logger = logging.getLogger(__name__)
                    logger.info(
                                        f"[{base_slot}]  Calibration:  {method_id}  ?
{calibration_result.final_score:.3f}"
                    )

                except MethodBelowThresholdError as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    logger.error(
                        f"[{base_slot}] Method {method_id} FAILED calibration: "
                        f"score={e.score:.3f}, threshold={e.threshold:.3f}"
                    )
                    raise RuntimeError(
                        f"Method {method_id} failed calibration threshold"
                    ) from e
                except Exception as e:
                    import logging
                    logger = logging.getLogger(__name__)
                       logger.warning(f"[{base_slot}] Calibration error for {method_id}:
{e}")

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **common_kwargs,
            )
            method_outputs["primary_analysis"] = result

            # Track signal usage
            if signal_pack is not None:
                signal_usage_list.append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": signal_pack.policy_area,
                        "version": signal_pack.version,
                    }
                )

        # Store signal usage in method_outputs for trace
        if signal_usage_list:
            method_outputs["_signal_usage"] = signal_usage_list

        # NEW: Evidence assembly and validation using EvidenceNexus
         # Note: EvidenceNexus extracts assembly_rules and validation_rules from contract
directly
        validation_rules_section = contract.get("validation_rules", {})
```

```
        nexus_result = process_evidence(
            method_outputs=method_outputs,
            question_context={
                "question_id": question_id,
                "question_global": question_global,
                "policy_area_id": policy_area_id,
                "dimension_id": dimension_id,
                "expected_elements": expected_elements,
                "patterns": patterns,
                       # Provide raw text so EvidenceNexus can run pattern extraction
deterministically.
                "raw_text": getattr(document, "raw_text", "") or "",
                # Provide document context for scope coherence + context filters.
                "document_context": document_context,
                # Provide SISAS consumption tracker for proof + utilization metrics.
                "consumption_tracker": consumption_tracker,
            },
            signal_pack=signal_pack,  # SISAS: Enable signal provenance
            contract=contract,
        )

        evidence = nexus_result["evidence"]
        trace = nexus_result["trace"]
        validation = nexus_result["validation"]

        # Get error_handling for subsequent validation orchestrator
        error_handling = contract.get("error_handling", {})

                      #   Reconstruct   validation_rules_object   for   compatibility   with
ValidationOrchestrator
        validation_rules = validation_rules_section.get("rules", [])
        na_policy = validation_rules_section.get("na_policy", "abort_on_critical")
        validation_rules_object = {"rules": validation_rules, "na_policy": na_policy}

        # CONTRACT VALIDATION with ValidationOrchestrator
        contract_validation = None
        if self._use_validation_orchestrator:
                                                                                    from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator
import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules_object,
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
```

```
            orchestrator=self.validation_orchestrator,
            auto_register=True
        )

        # Merge contract validation failures into standard validation
        if not contract_validation.passed:
            validation["contract_validation_failed"] = True
            validation["contract_error_code"] = contract_validation.error_code
            validation["contract_remediation"] = contract_validation.remediation
            validation["contract_failures"] = [
                {
                    "type": f.failure_type,
                    "field": f.field_name,
                    "message": f.message,
                    "severity": f.severity
                }
                for f in contract_validation.failures_detailed[:10]
            ]

    # Handle validation failures based on NA policy
    validation_passed = validation.get("passed", True)
    if not validation_passed:
        if na_policy == "abort_on_critical":
            # Error handling will check failure contract below
            pass  # Let error_handling section handle abort
        elif na_policy == "score_zero":
            # Mark result as failed with score zero
            validation["score"] = 0.0
            validation["quality_level"] = "FAILED_VALIDATION"
            validation["na_policy_applied"] = "score_zero"
        elif na_policy == "propagate":
            # Continue with validation errors in result
            validation["na_policy_applied"] = "propagate"
            validation["validation_failed"] = True

    # Error handling
    error_handling = contract["error_handling"]
    if error_handling:
        evidence_with_validation = {**evidence, "validation": validation}
        self._check_failure_contract(evidence_with_validation, error_handling)

    # Build result
    result_data = {
        "base_slot": base_slot,
        "question_id": question_id,
        "question_global": question_global,
        "policy_area_id": policy_area_id,
        "dimension_id": dimension_id,
        "cluster_id": identity.get("cluster_id"),
        "evidence": evidence,
        "validation": validation,
        "trace": trace,
        # CONTRACT VALIDATION METADATA
        "contract_validation": {
```

```
                "enabled": contract_validation is not None,
                "passed": contract_validation.passed if contract_validation else None,
                 "error_code": contract_validation.error_code if contract_validation else
None,
                          "failure_count":  len(contract_validation.failures_detailed)  if
contract_validation else 0,
                "orchestrator_registered": self._use_validation_orchestrator
            },
            # CALIBRATION METADATA
            "calibration_metadata": {
                "enabled": self.calibration_orchestrator is not None,
                "results": calibration_results,
                "summary": {
                    "total_methods": len(calibration_results),
                    "average_score": sum(
                        cr["final_score"] for cr in calibration_results.values()
                    ) / len(calibration_results) if calibration_results else 0.0,
                    "min_score": min(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                    "max_score": max(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                }
            }
        }

         # NEW: Record evidence provenance in EvidenceNexus internal store (if available
in nexus_result)
        if "provenance_record" in nexus_result:
            # Provenance is now handled internally by EvidenceNexus
            result_data["provenance"] = nexus_result["provenance_record"]

        # NEW: Add EvidenceNexus scoring fields for Phase 3
        # These are essential for Phase 3 to extract scores for aggregation
        if "overall_confidence" in nexus_result:
            result_data["overall_confidence"] = nexus_result["overall_confidence"]
        if "completeness" in nexus_result:
            result_data["completeness"] = nexus_result["completeness"]
        if "calibrated_interval" in nexus_result:
            result_data["calibrated_interval"] = nexus_result["calibrated_interval"]
        if "synthesized_answer" in nexus_result:
            result_data["synthesized_answer"] = nexus_result["synthesized_answer"]

        # Validate output against output_contract schema if present
        output_contract = contract.get("output_contract", {})
        if output_contract and "schema" in output_contract:
            self._validate_output_contract(
                result_data, output_contract["schema"], base_slot
            )

        # NEW: Generate doctoral-level narrative using Carver synthesizer
```

```
        human_readable_config = output_contract.get("human_readable_output", {})
        if human_readable_config or nexus_result.get("graph"):
            # Use Carver to generate PhD-level narrative from evidence graph
            carver = DoctoralCarverSynthesizer()

            try:
                # Use synthesize_structured for full object access
                carver_answer = carver.synthesize_structured(evidence, contract)
                result_data["human_readable_output"] = carver_answer.to_human_readable()
                if hasattr(carver_answer, "synthesis_trace"):
                    result_data["carver_metrics"] = carver_answer.synthesis_trace
            except Exception as e:
                import logging
                logging.error(f"Carver synthesis failed: {e}", exc_info=True)
                # Fallback to basic template if Carver fails
                                                result_data["human_readable_output"]   =
self._generate_human_readable_output(
                    evidence, validation, human_readable_config, contract
                )
                result_data["carver_error"] = str(e)

        return result_data

    def _validate_output_contract(
        self, result: dict[str, Any], schema: dict[str, Any], base_slot: str
    ) -> None:
        """Validate result against output_contract schema with detailed error messages.

        Args:
            result: Result data to validate
            schema: JSON Schema from contract
            base_slot: Base slot identifier for error messages

        Raises:
            ValueError: If validation fails with detailed path information
        """
        from jsonschema import ValidationError, validate

        try:
            validate(instance=result, schema=schema)
        except ValidationError as e:
            # Enhanced error message with JSON path
            path = (
                ".".join(str(p) for p in e.absolute_path) if e.absolute_path else "root"
            )
            raise ValueError(
                      f"Output contract validation failed for {base_slot} at '{path}':
{e.message}. "
                f"Schema constraint: {e.schema}"
            ) from e

    def _generate_human_readable_output(
        self,
        evidence: dict[str, Any],
```

```
    validation: dict[str, Any],
    config: dict[str, Any],
    contract: dict[str, Any],
) -> str:
    """Generate production-grade human-readable output from template.

    Implements full template engine with:
    - Variable substitution with dot-notation: {evidence.elements_found_count}
    - Derived metrics: Automatic calculation of means, counts, percentages
    - List formatting: Convert arrays to markdown/html/plain_text lists
    - Methodological depth rendering: Full epistemological documentation
    - Multi-format support: markdown, html, plain_text with proper formatting

    Args:
        evidence: Evidence dict from executor
        validation: Validation dict
        config: human_readable_output config from contract
        contract: Full contract for methodological_depth access

    Returns:
        Formatted string in specified format
    """
    template_config = config.get("template", {})
    format_type = config.get("format", "markdown")
    methodological_depth_config = config.get("methodological_depth", {})

    # Build context for variable substitution
    context = self._build_template_context(evidence, validation, contract)

    # Render each template section
    sections = []

    # Title
    if "title" in template_config:
        sections.append(
            self._render_template_string(
                template_config["title"], context, format_type
            )
        )

    # Summary
    if "summary" in template_config:
        sections.append(
            self._render_template_string(
                template_config["summary"], context, format_type
            )
        )

    # Score section
    if "score_section" in template_config:
        sections.append(
            self._render_template_string(
                template_config["score_section"], context, format_type
            )
```

```
        )

    # Elements section
    if "elements_section" in template_config:
        sections.append(
            self._render_template_string(
                template_config["elements_section"], context, format_type
            )
        )

    # Details (list of items)
    if "details" in template_config and isinstance(
        template_config["details"], list
    ):
        detail_items = [
            self._render_template_string(item, context, format_type)
            for item in template_config["details"]
        ]
        sections.append(self._format_list(detail_items, format_type))

    # Interpretation
    if "interpretation" in template_config:
        # Add methodological interpretation if available
        context["methodological_interpretation"] = (
            self._render_methodological_depth(
                methodological_depth_config, evidence, validation, format_type
            )
        )
        sections.append(
            self._render_template_string(
                template_config["interpretation"], context, format_type
            )
        )

    # Recommendations
    if "recommendations" in template_config:
        sections.append(
            self._render_template_string(
                template_config["recommendations"], context, format_type
            )
        )

    # Join sections with appropriate separator for format
    separator = (
        "\n\n"
        if format_type == "markdown"
        else "\n\n" if format_type == "plain_text" else "<br><br>"
    )
    return separator.join(filter(None, sections))

def _build_template_context(
    self,
    evidence: dict[str, Any],
    validation: dict[str, Any],
```

```
    contract: dict[str, Any],
) -> dict[str, Any]:
    """Build comprehensive context for template variable substitution.

    Args:
        evidence: Evidence dict
        validation: Validation dict
        contract: Full contract

    Returns:
        Context dict with all variables and derived metrics
    """
    # Base context
    context = {
        "evidence": evidence.copy(),
        "validation": validation.copy(),
    }

    # Add derived metrics from evidence
    if "elements" in evidence and isinstance(evidence["elements"], list):
        context["evidence"]["elements_found_count"] = len(evidence["elements"])
        context["evidence"]["elements_found_list"] = self._format_evidence_list(
            evidence["elements"]
        )

    if "confidences" in evidence and isinstance(evidence["confidences"], list):
        confidences = evidence["confidences"]
        if confidences:
            context["evidence"]["confidence_scores"] = {
                "mean": sum(confidences) / len(confidences),
                "min": min(confidences),
                "max": max(confidences),
            }

    if "patterns" in evidence and isinstance(evidence["patterns"], dict):
        context["evidence"]["pattern_matches_count"] = len(evidence["patterns"])

    # Add defaults for missing keys to prevent KeyError
    context["evidence"].setdefault("missing_required_elements", "None")
    context["evidence"].setdefault("official_sources_count", 0)
    context["evidence"].setdefault("quantitative_indicators_count", 0)
    context["evidence"].setdefault("temporal_series_count", 0)
    context["evidence"].setdefault("territorial_coverage", "Not specified")
    context["evidence"].setdefault(
        "recommendations", "No specific recommendations available"
    )

    # Add score and quality from validation or defaults
    context["score"] = validation.get("score", 0.0)
    context["quality_level"] = self._determine_quality_level(
        validation.get("score", 0.0)
    )

    return context
```

```
def _determine_quality_level(self, score: float) -> str:
    """Determine quality level from score.

    Args:
        score: Numeric score (typically 0.0-3.0)

    Returns:
        Quality level string
    """
    if score >= 2.5:
        return "EXCELLENT"
    elif score >= 2.0:
        return "GOOD"
    elif score >= 1.0:
        return "ACCEPTABLE"
    elif score > 0:
        return "INSUFFICIENT"
    else:
        return "FAILED"

def _render_template_string(
    self, template: str, context: dict[str, Any], format_type: str
) -> str:
    """Render a template string with variable substitution.

    Supports dot-notation: {evidence.elements_found_count}
    Supports arithmetic: {score}/3.0 (rendered as-is, user interprets)

    Args:
        template: Template string with {variable} placeholders
        context: Context dict
        format_type: Output format (markdown, html, plain_text)

    Returns:
        Rendered string with variables substituted
    """
    import re

    def replace_var(match):
        var_path = match.group(1)
        try:
            # Handle dot-notation traversal
            keys = var_path.split(".")
            value = context
            for key in keys:
                if isinstance(value, dict):
                    value = value[key]
                else:
                    # Try to get attribute (for objects)
                    value = getattr(value, key, None)
                    if value is None:
                        return f"{{{MISSING:{var_path}}}}"
```

```
            # Format value appropriately
            if isinstance(value, float):
                return f"{value:.2f}"
            elif isinstance(value, list | dict):
                return str(value)  # Simple representation
            else:
                return str(value)
        except (KeyError, AttributeError, TypeError):
            return f"{{MISSING:{var_path}}}"

    # Replace all {variable} patterns
    rendered = re.sub(r"\{([^}]+)\}", replace_var, template)
    return rendered

def _format_evidence_list(self, elements: list) -> str:
    """Format evidence elements as markdown list.

    Args:
        elements: List of evidence elements

    Returns:
        Markdown-formatted list string
    """
    if not elements:
        return "- No elements found"

    formatted = []
    for elem in elements:
        if isinstance(elem, dict):
            # Try to extract meaningful representation
            elem_str = elem.get("description") or elem.get("type") or str(elem)
        else:
            elem_str = str(elem)
        formatted.append(f"- {elem_str}")

    return "\n".join(formatted)

def _format_list(self, items: list[str], format_type: str) -> str:
    """Format a list of items according to output format.

    Args:
        items: List of string items
        format_type: Output format

    Returns:
        Formatted list string
    """
    if format_type == "html":
        items_html = "".join(f"<li>{item}</li>" for item in items)
        return f"<ul>{items_html}</ul>"
    else:  # markdown or plain_text
        return "\n".join(f"- {item}" for item in items)

def _render_methodological_depth(
```

```
        self,
        config: dict[str, Any],
        evidence: dict[str, Any],
        validation: dict[str, Any],
        format_type: str,
    ) -> str:
        """Render methodological depth section with epistemological foundations.

        Transforms v3 contract's methodological_depth into comprehensive documentation.

        Args:
            config: methodological_depth config from contract
            evidence: Evidence dict for contextualization
            validation: Validation dict
            format_type: Output format

        Returns:
            Formatted methodological depth documentation
        """
        if not config or "methods" not in config:
            return "Methodological documentation not available for this executor."

        sections = []

        # Header
        if format_type == "markdown":
            sections.append("#### Methodological Foundations\n")
        elif format_type == "html":
            sections.append("<h4>Methodological Foundations</h4>")
        else:
            sections.append("METHODOLOGICAL FOUNDATIONS\n")

        methods = config.get("methods", [])

        for method_info in methods:
            method_name = method_info.get("method_name", "Unknown")
            class_name = method_info.get("class_name", "Unknown")
            priority = method_info.get("priority", 0)
            role = method_info.get("role", "analysis")

            # Method header
            if format_type == "markdown":
                sections.append(
                          f"##### {class_name}.{method_name} (Priority {priority}, Role:
{role})\n"
                )
            else:
                sections.append(
                              f"\n{class_name}.{method_name} (Priority {priority}, Role:
{role})\n"
                )

            # Epistemological foundation
            epist = method_info.get("epistemological_foundation", {})
```

```
        if epist:
            sections.append(
                self._render_epistemological_foundation(epist, format_type)
            )

        # Technical approach
        technical = method_info.get("technical_approach", {})
        if technical:
            sections.append(self._render_technical_approach(technical, format_type))

        # Output interpretation
        output_interp = method_info.get("output_interpretation", {})
        if output_interp:
            sections.append(
                self._render_output_interpretation(output_interp, format_type)
            )

    # Method combination logic
    combination = config.get("method_combination_logic", {})
    if combination:
        sections.append(self._render_method_combination(combination, format_type))

    return "\n\n".join(filter(None, sections))

def _render_epistemological_foundation(
    self, foundation: dict[str, Any], format_type: str
) -> str:
    """Render epistemological foundation section.

    Args:
        foundation: Epistemological foundation dict
        format_type: Output format

    Returns:
        Formatted epistemological foundation text
    """
    parts = []

    paradigm = foundation.get("paradigm")
    if paradigm:
        parts.append(f"**Paradigm**: {paradigm}")

    ontology = foundation.get("ontological_basis")
    if ontology:
        parts.append(f"**Ontological Basis**: {ontology}")

    stance = foundation.get("epistemological_stance")
    if stance:
        parts.append(f"**Epistemological Stance**: {stance}")

    framework = foundation.get("theoretical_framework", [])
    if framework:
        parts.append("**Theoretical Framework**:")
        for item in framework:
```

```
            parts.append(f"  - {item}")

    justification = foundation.get("justification")
    if justification:
        parts.append(f"**Justification**: {justification}")

    return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

def _render_technical_approach(
    self, technical: dict[str, Any], format_type: str
) -> str:
    """Render technical approach section.

    Args:
        technical: Technical approach dict
        format_type: Output format

    Returns:
        Formatted technical approach text
    """
    parts = []

    method_type = technical.get("method_type")
    if method_type:
        parts.append(f"**Method Type**: {method_type}")

    algorithm = technical.get("algorithm")
    if algorithm:
        parts.append(f"**Algorithm**: {algorithm}")

    steps = technical.get("steps", [])
    if steps:
        parts.append("**Processing Steps**:")
        for step in steps:
            step_num = step.get("step", "?")
            step_name = step.get("name", "Unnamed")
            step_desc = step.get("description", "")
            parts.append(f"  {step_num}. **{step_name}**: {step_desc}")

    assumptions = technical.get("assumptions", [])
    if assumptions:
        parts.append("**Assumptions**:")
        for assumption in assumptions:
            parts.append(f"  - {assumption}")

    limitations = technical.get("limitations", [])
    if limitations:
        parts.append("**Limitations**:")
        for limitation in limitations:
            parts.append(f"  - {limitation}")

    return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

def _render_output_interpretation(
```

```
    self, interpretation: dict[str, Any], format_type: str
) -> str:
    """Render output interpretation section.

    Args:
        interpretation: Output interpretation dict
        format_type: Output format

    Returns:
        Formatted output interpretation text
    """
    parts = []

    guide = interpretation.get("interpretation_guide", {})
    if guide:
        parts.append("**Interpretation Guide**:")
        for threshold_name, threshold_desc in guide.items():
            parts.append(f"  - **{threshold_name}**: {threshold_desc}")

    insights = interpretation.get("actionable_insights", [])
    if insights:
        parts.append("**Actionable Insights**:")
        for insight in insights:
            parts.append(f"  - {insight}")

    return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

def _render_method_combination(
    self, combination: dict[str, Any], format_type: str
) -> str:
    """Render method combination logic section.

    Args:
        combination: Method combination dict
        format_type: Output format

    Returns:
        Formatted method combination text
    """
    parts = []

    if format_type == "markdown":
        parts.append("#### Method Combination Strategy\n")
    else:
        parts.append("METHOD COMBINATION STRATEGY\n")

    strategy = combination.get("combination_strategy")
    if strategy:
        parts.append(f"**Strategy**: {strategy}")

    rationale = combination.get("rationale")
    if rationale:
        parts.append(f"**Rationale**: {rationale}")
```

```
fusion = combination.get("evidence_fusion")
if fusion:
    parts.append(f"**Evidence Fusion**: {fusion}")

return "\n".join(parts) if format_type != "html" else "<br>".join(parts)
```

```
src/farfan_pipeline/phases/Phase_two/batch_executor.py

"""Batch processing infrastructure for executor scalability.

This module provides batched entity processing with:
- Configurable batch sizes per executor type based on object complexity
- Streaming result aggregation to avoid memory accumulation
- Async executor flow integration for parallel batch processing
- Batch-level error handling with partial success recovery

Example:
    >>> config = BatchExecutorConfig(default_batch_size=10, max_batch_size=100)
    >>> executor = BatchExecutor(config, method_executor, signal_registry)
    >>> async for batch_result in executor.execute_batches(entities, question_context):
    ...     process_batch_result(batch_result)
"""

from __future__ import annotations

import asyncio
import logging
import time
from collections.abc import AsyncIterator, Callable, Iterable
from dataclasses import dataclass, field
from enum import Enum
from typing import TYPE_CHECKING, Any, TypedDict

if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor
    from farfan_pipeline.core.types import PreprocessedDocument

logger = logging.getLogger(__name__)


class BatchStatus(Enum):
    """Status of batch execution."""

    PENDING = "pending"
    RUNNING = "running"
    COMPLETED = "completed"
    PARTIAL_SUCCESS = "partial_success"
    FAILED = "failed"
    CANCELLED = "cancelled"


class ExecutorComplexity(Enum):
    """Complexity classification for executor types."""

    SIMPLE = "simple"
    MODERATE = "moderate"
    COMPLEX = "complex"
    VERY_COMPLEX = "very_complex"
```

```
@dataclass
class BatchExecutorConfig:
    """Configuration for batch processing.

    Attributes:
        default_batch_size: Default batch size for unclassified executors
        max_batch_size: Maximum batch size allowed
        min_batch_size: Minimum batch size allowed
        enable_streaming: Enable streaming result aggregation
        error_threshold: Fraction of failures before marking batch as failed (0.0-1.0)
        max_retries: Maximum number of retries for failed batches
        backoff_base_seconds: Base delay for exponential backoff
        enable_instrumentation: Enable detailed logging and metrics
    """

    default_batch_size: int = 10
    max_batch_size: int = 100
    min_batch_size: int = 1
    enable_streaming: bool = True
    error_threshold: float = 0.5
    max_retries: int = 2
    backoff_base_seconds: float = 1.0
    enable_instrumentation: bool = True

    def __post_init__(self) -> None:
        """Validate configuration parameters."""
        if not (
            1 <= self.min_batch_size <= self.default_batch_size <= self.max_batch_size
        ):
            raise ValueError(
                f"Invalid batch size configuration: min={self.min_batch_size}, "
                f"default={self.default_batch_size}, max={self.max_batch_size}"
            )
        if not 0.0 <= self.error_threshold <= 1.0:
            raise ValueError(
                f"error_threshold must be in [0.0, 1.0], got {self.error_threshold}"
            )


EXECUTOR_COMPLEXITY_MAP: dict[str, ExecutorComplexity] = {
    "D1-Q1": ExecutorComplexity.MODERATE,
    "D1-Q2": ExecutorComplexity.MODERATE,
    "D1-Q3": ExecutorComplexity.COMPLEX,
    "D1-Q4": ExecutorComplexity.MODERATE,
    "D1-Q5": ExecutorComplexity.MODERATE,
    "D2-Q1": ExecutorComplexity.MODERATE,
    "D2-Q2": ExecutorComplexity.VERY_COMPLEX,
    "D2-Q3": ExecutorComplexity.COMPLEX,
    "D2-Q4": ExecutorComplexity.COMPLEX,
    "D2-Q5": ExecutorComplexity.MODERATE,
    "D3-Q1": ExecutorComplexity.MODERATE,
    "D3-Q2": ExecutorComplexity.VERY_COMPLEX,
    "D3-Q3": ExecutorComplexity.VERY_COMPLEX,
    "D3-Q4": ExecutorComplexity.VERY_COMPLEX,
```

```
    "D3-Q5": ExecutorComplexity.VERY_COMPLEX,
    "D4-Q1": ExecutorComplexity.VERY_COMPLEX,
    "D4-Q2": ExecutorComplexity.COMPLEX,
    "D4-Q3": ExecutorComplexity.COMPLEX,
    "D4-Q4": ExecutorComplexity.MODERATE,
    "D4-Q5": ExecutorComplexity.MODERATE,
    "D5-Q1": ExecutorComplexity.MODERATE,
    "D5-Q2": ExecutorComplexity.VERY_COMPLEX,
    "D5-Q3": ExecutorComplexity.MODERATE,
    "D5-Q4": ExecutorComplexity.COMPLEX,
    "D5-Q5": ExecutorComplexity.COMPLEX,
    "D6-Q1": ExecutorComplexity.COMPLEX,
    "D6-Q2": ExecutorComplexity.MODERATE,
    "D6-Q3": ExecutorComplexity.COMPLEX,
    "D6-Q4": ExecutorComplexity.MODERATE,
    "D6-Q5": ExecutorComplexity.MODERATE,
}

COMPLEXITY_BATCH_SIZE_MAP: dict[ExecutorComplexity, int] = {
    ExecutorComplexity.SIMPLE: 50,
    ExecutorComplexity.MODERATE: 20,
    ExecutorComplexity.COMPLEX: 10,
    ExecutorComplexity.VERY_COMPLEX: 5,
}


class BatchMetrics(TypedDict):
    """Metrics for a single batch execution."""

    batch_id: str
    batch_index: int
    batch_size: int
    status: BatchStatus
    start_time: float
    end_time: float | None
    execution_time_ms: float
    successful_items: int
    failed_items: int
    retries_used: int
    error_messages: list[str]


@dataclass
class BatchResult:
    """Result of a batch execution.

    Attributes:
        batch_id: Unique batch identifier
        batch_index: Index of the batch in the sequence
        items: List of items in this batch
        results: List of results corresponding to items (may contain None for failures)
        status: Batch execution status
        metrics: Execution metrics
        errors: List of errors encountered (empty if successful)
```

```
    """

    batch_id: str
    batch_index: int
    items: list[Any]
    results: list[Any]
    status: BatchStatus
    metrics: BatchMetrics
    errors: list[dict[str, Any]] = field(default_factory=list)

    def is_successful(self) -> bool:
        """Check if batch execution was fully successful."""
        return self.status == BatchStatus.COMPLETED and not self.errors

    def has_partial_success(self) -> bool:
        """Check if batch had partial success."""
        return self.status == BatchStatus.PARTIAL_SUCCESS and any(
            r is not None for r in self.results
        )

    def get_successful_results(self) -> list[tuple[Any, Any]]:
        """Get list of (item, result) pairs for successful executions."""
        return [
            (item, result)
            for item, result in zip(self.items, self.results, strict=False)
            if result is not None
        ]

    def get_failed_items(self) -> list[Any]:
        """Get list of items that failed execution."""
        return [
            item
            for item, result in zip(self.items, self.results, strict=False)
            if result is None
        ]


@dataclass
class AggregatedBatchResults:
    """Aggregated results from multiple batch executions.

    Attributes:
        total_batches: Total number of batches processed
        total_items: Total number of items processed
        successful_items: Number of successfully processed items
        failed_items: Number of failed items
        results: List of all successful results
        errors: List of all errors encountered
        execution_time_ms: Total execution time in milliseconds
        batch_metrics: List of metrics for each batch
    """

    total_batches: int
    total_items: int
```

```
    successful_items: int
    failed_items: int
    results: list[Any]
    errors: list[dict[str, Any]]
    execution_time_ms: float
    batch_metrics: list[BatchMetrics]

    def success_rate(self) -> float:
        """Calculate success rate as fraction of successful items."""
        if self.total_items == 0:
            return 0.0
        return self.successful_items / self.total_items


class BatchExecutor:
    """Batch processing executor with streaming aggregation and error recovery.

    This executor provides scalable batch processing for entity collections with:
    - Adaptive batch sizing based on executor complexity
    - Streaming result aggregation to avoid memory accumulation
    - Parallel batch processing via async executor flow
    - Batch-level error handling with partial success recovery
    """

    def __init__(
        self,
        config: BatchExecutorConfig | None = None,
        method_executor: MethodExecutor | None = None,
        signal_registry: Any | None = None,
        questionnaire_provider: Any | None = None,
        calibration_orchestrator: Any | None = None,
    ) -> None:
        """Initialize batch executor.

        Args:
            config: Batch execution configuration
            method_executor: MethodExecutor instance for method routing
            signal_registry: Signal registry for executor instances
            questionnaire_provider: Questionnaire provider
            calibration_orchestrator: Calibration orchestrator
        """
        self.config = config or BatchExecutorConfig()
        self.method_executor = method_executor
        self.signal_registry = signal_registry
        self.questionnaire_provider = questionnaire_provider
        self.calibration_orchestrator = calibration_orchestrator
        self._batch_counter = 0

        if self.config.enable_instrumentation:
            logger.info(
                                                    f"BatchExecutor    initialized:
default_batch_size={self.config.default_batch_size}, "
                f"max_batch_size={self.config.max_batch_size}, "
                f"error_threshold={self.config.error_threshold}"
```

```
        )

def get_batch_size_for_executor(self, base_slot: str) -> int:
    """Determine batch size for a given executor based on complexity.

    Args:
        base_slot: Executor base slot (e.g., "D1-Q1")

    Returns:
        Recommended batch size for this executor
    """
    complexity = EXECUTOR_COMPLEXITY_MAP.get(base_slot, ExecutorComplexity.MODERATE)
    recommended_size = COMPLEXITY_BATCH_SIZE_MAP.get(
        complexity, self.config.default_batch_size
    )

    batch_size = max(
        self.config.min_batch_size,
        min(recommended_size, self.config.max_batch_size),
    )

    if self.config.enable_instrumentation:
        logger.debug(
            f"Batch size for {base_slot}: {batch_size} "
            f"(complexity={complexity.value}, recommended={recommended_size})"
        )

    return batch_size

def _create_batches(
    self, items: list[Any], batch_size: int
) -> list[tuple[int, list[Any]]]:
    """Split items into batches of specified size.

    Args:
        items: List of items to batch
        batch_size: Size of each batch

    Returns:
        List of (batch_index, batch_items) tuples
    """
    batches = []
    for i in range(0, len(items), batch_size):
        batch_index = i // batch_size
        batch_items = items[i : i + batch_size]
        batches.append((batch_index, batch_items))

    if self.config.enable_instrumentation:
        logger.info(
            f"Created {len(batches)} batches from {len(items)} items "
            f"(batch_size={batch_size})"
        )

    return batches
```

```
    async def _execute_single_batch(
        self,
        batch_id: str,
        batch_index: int,
        batch_items: list[Any],
        executor_instance: Any,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
    ) -> BatchResult:
        """Execute a single batch of items.

        Args:
            batch_id: Unique batch identifier
            batch_index: Index of this batch
            batch_items: Items to process in this batch
            executor_instance: Executor instance to use
            document: Document being processed
            question_context: Question context for execution

        Returns:
            BatchResult with execution details
        """
        start_time = time.perf_counter()
        results: list[Any] = []
        errors: list[dict[str, Any]] = []
        successful_items = 0
        failed_items = 0

        if self.config.enable_instrumentation:
            logger.debug(
                    f"[{batch_id}] Executing batch {batch_index} with {len(batch_items)}
items"
            )

        for item_index, item in enumerate(batch_items):
            try:
                result = await asyncio.to_thread(
                    executor_instance.execute,
                    document,
                    self.method_executor,
                    question_context=question_context,
                )
                results.append(result)
                successful_items += 1
            except Exception as exc:
                results.append(None)
                failed_items += 1
                error_detail = {
                    "batch_id": batch_id,
                    "batch_index": batch_index,
                    "item_index": item_index,
                    "error_type": type(exc).__name__,
                    "error_message": str(exc),
```

```
            }
            errors.append(error_detail)

            if self.config.enable_instrumentation:
                logger.warning(
                    f"[{batch_id}] Item {item_index} failed: {exc}", exc_info=False
                )

    end_time = time.perf_counter()
    execution_time_ms = (end_time - start_time) * 1000.0

    error_rate = failed_items / len(batch_items) if batch_items else 0.0
    if error_rate >= self.config.error_threshold:
        status = BatchStatus.FAILED
    elif failed_items > 0:
        status = BatchStatus.PARTIAL_SUCCESS
    else:
        status = BatchStatus.COMPLETED

    metrics: BatchMetrics = {
        "batch_id": batch_id,
        "batch_index": batch_index,
        "batch_size": len(batch_items),
        "status": status,
        "start_time": start_time,
        "end_time": end_time,
        "execution_time_ms": execution_time_ms,
        "successful_items": successful_items,
        "failed_items": failed_items,
        "retries_used": 0,
        "error_messages": [e["error_message"] for e in errors],
    }

    if self.config.enable_instrumentation:
        logger.info(
            f"[{batch_id}] Batch {batch_index} completed: "
            f"{successful_items}/{len(batch_items)} successful "
            f"({execution_time_ms:.2f}ms, status={status.value})"
        )

    return BatchResult(
        batch_id=batch_id,
        batch_index=batch_index,
        items=batch_items,
        results=results,
        status=status,
        metrics=metrics,
        errors=errors,
    )

async def _execute_batch_with_retry(
    self,
    batch_id: str,
    batch_index: int,
```

```
        batch_items: list[Any],
        executor_instance: Any,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
    ) -> BatchResult:
        """Execute a batch with retry logic for failed batches.

        Args:
            batch_id: Unique batch identifier
            batch_index: Index of this batch
            batch_items: Items to process in this batch
            executor_instance: Executor instance to use
            document: Document being processed
            question_context: Question context for execution

        Returns:
            BatchResult with execution details
        """
        retry_count = 0
        last_result: BatchResult | None = None

        while retry_count <= self.config.max_retries:
            result = await self._execute_single_batch(
                batch_id,
                batch_index,
                batch_items,
                executor_instance,
                document,
                question_context,
            )

            if result.status in (BatchStatus.COMPLETED, BatchStatus.PARTIAL_SUCCESS):
                result.metrics["retries_used"] = retry_count
                return result

            last_result = result
            retry_count += 1

            if retry_count <= self.config.max_retries:
                backoff_delay = self.config.backoff_base_seconds * (
                    2 ** (retry_count - 1)
                )
                if self.config.enable_instrumentation:
                    logger.warning(
                                    f"[{batch_id}] Batch {batch_index} failed (attempt
{retry_count}), "
                        f"retrying after {backoff_delay:.2f}s"
                    )
                await asyncio.sleep(backoff_delay)

        if last_result:
            last_result.metrics["retries_used"] = retry_count - 1
            if self.config.enable_instrumentation:
                logger.error(
```

```
                    f"[{batch_id}] Batch {batch_index} failed after {retry_count - 1}
retries"
                )
            return last_result

        raise RuntimeError(f"Batch {batch_id} execution failed without result")

    async def execute_batches(
        self,
        items: list[Any],
        executor_class: type,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
        base_slot: str | None = None,
        batch_size: int | None = None,
    ) -> AsyncIterator[BatchResult]:
        """Execute items in batches with streaming result generation.

        This method processes items in batches and yields results as they complete,
        enabling streaming aggregation without accumulating all results in memory.

        Args:
            items: List of items to process
            executor_class: Executor class to instantiate
            document: Document being processed
            question_context: Question context for execution
            base_slot: Base slot for complexity-based batch sizing (optional)
              batch_size: Override batch size (optional, uses complexity-based sizing if
None)

        Yields:
            BatchResult for each completed batch

        Example:
            >>> batches = executor.execute_batches(entities, D1Q1_Executor, doc, ctx)
            >>> async for batch_result in batches:
            ...     for item, result in batch_result.get_successful_results():
            ...         aggregate(result)
        """
        if not items:
            logger.warning("execute_batches called with empty items list")
            return

        if batch_size is None:
            if base_slot:
                batch_size = self.get_batch_size_for_executor(base_slot)
            else:
                batch_size = self.config.default_batch_size

        batches = self._create_batches(items, batch_size)

        for batch_index, batch_items in batches:
            self._batch_counter += 1
            batch_id = f"batch_{self._batch_counter:06d}"
```

```
        executor_instance = executor_class(
            method_executor=self.method_executor,
            signal_registry=self.signal_registry,
            config=self.config,
            questionnaire_provider=self.questionnaire_provider,
            calibration_orchestrator=self.calibration_orchestrator,
        )

        batch_result = await self._execute_batch_with_retry(
            batch_id,
            batch_index,
            batch_items,
            executor_instance,
            document,
            question_context,
        )

        yield batch_result

async def execute_batches_parallel(
    self,
    items: list[Any],
    executor_class: type,
    document: PreprocessedDocument,
    question_context: dict[str, Any],
    base_slot: str | None = None,
    batch_size: int | None = None,
    max_concurrent_batches: int = 4,
) -> list[BatchResult]:
    """Execute batches in parallel with concurrency control.

    Args:
        items: List of items to process
        executor_class: Executor class to instantiate
        document: Document being processed
        question_context: Question context for execution
        base_slot: Base slot for complexity-based batch sizing (optional)
        batch_size: Override batch size (optional)
        max_concurrent_batches: Maximum number of concurrent batch executions

    Returns:
        List of BatchResults for all batches
    """
    if not items:
        return []

    if batch_size is None:
        if base_slot:
            batch_size = self.get_batch_size_for_executor(base_slot)
        else:
            batch_size = self.config.default_batch_size

    batches = self._create_batches(items, batch_size)
```

```
    semaphore = asyncio.Semaphore(max_concurrent_batches)
    results: list[BatchResult] = []

    async def process_batch_with_semaphore(
        batch_index: int, batch_items: list[Any]
    ) -> BatchResult:
        async with semaphore:
            self._batch_counter += 1
            batch_id = f"batch_{self._batch_counter:06d}"

            executor_instance = executor_class(
                method_executor=self.method_executor,
                signal_registry=self.signal_registry,
                config=self.config,
                questionnaire_provider=self.questionnaire_provider,
                calibration_orchestrator=self.calibration_orchestrator,
            )

            return await self._execute_batch_with_retry(
                batch_id,
                batch_index,
                batch_items,
                executor_instance,
                document,
                question_context,
            )

    tasks = [
        asyncio.create_task(process_batch_with_semaphore(batch_index, batch_items))
        for batch_index, batch_items in batches
    ]

    if self.config.enable_instrumentation:
        logger.info(
            f"Executing {len(tasks)} batches in parallel "
            f"(max_concurrent={max_concurrent_batches})"
        )

    for task in asyncio.as_completed(tasks):
        result = await task
        results.append(result)

    return results

async def aggregate_batch_results(
    self, batch_results: Iterable[BatchResult]
) -> AggregatedBatchResults:
    """Aggregate results from multiple batches.

    Args:
        batch_results: Iterable of BatchResults to aggregate

    Returns:
        AggregatedBatchResults with summary statistics
```

```
        """
        start_time = time.perf_counter()

        total_batches = 0
        total_items = 0
        successful_items = 0
        failed_items = 0
        all_results: list[Any] = []
        all_errors: list[dict[str, Any]] = []
        batch_metrics_list: list[BatchMetrics] = []

        for batch_result in batch_results:
            total_batches += 1
            total_items += len(batch_result.items)
            successful_items += batch_result.metrics["successful_items"]
            failed_items += batch_result.metrics["failed_items"]

            for item, result in zip(
                batch_result.items, batch_result.results, strict=False
            ):
                if result is not None:
                    all_results.append(result)

            all_errors.extend(batch_result.errors)
            batch_metrics_list.append(batch_result.metrics)

        end_time = time.perf_counter()
        execution_time_ms = (end_time - start_time) * 1000.0

        aggregated = AggregatedBatchResults(
            total_batches=total_batches,
            total_items=total_items,
            successful_items=successful_items,
            failed_items=failed_items,
            results=all_results,
            errors=all_errors,
            execution_time_ms=execution_time_ms,
            batch_metrics=batch_metrics_list,
        )

        if self.config.enable_instrumentation:
            logger.info(
                  f"Aggregated {total_batches} batches: {successful_items}/{total_items}
successful "
                f"(success_rate={aggregated.success_rate():.2%}, "
                f"aggregation_time={execution_time_ms:.2f}ms)"
            )

        return aggregated

    async def stream_aggregate_batches(
        self,
        batch_stream: AsyncIterator[BatchResult],
        aggregation_fn: Callable[[Any, Any], Any],
```

```
) -> Any:
    """Stream aggregation of batch results without accumulating in memory.

    Args:
        batch_stream: Async iterator of BatchResults
        aggregation_fn: Function to aggregate results incrementally
            (accumulator, new_result) -> accumulator

    Returns:
        Final aggregated result

    Example:
        >>> def aggregate_fn(acc, result):
        ...     acc['count'] += 1
        ...     acc['total'] += result['score']
        ...     return acc
        >>>
        >>> batches = executor.execute_batches(entities, D1Q1_Executor, doc, ctx)
        >>> final = await executor.stream_aggregate_batches(batches, aggregate_fn)
    """
    accumulator = None

    async for batch_result in batch_stream:
        for result in batch_result.results:
            if result is not None:
                if accumulator is None:
                    accumulator = result
                else:
                    accumulator = aggregation_fn(accumulator, result)

    return accumulator
```

src/farfan_pipeline/phases/Phase_two/batch_generate_all_configs.py

"""
Batch generate all 30 executor config files with proper separation.

CRITICAL: These files contain ONLY runtime parameters (HOW).
NO calibration values (quality scores) are stored here.
"""

import json
from pathlib import Path

configs = {
    "D1_Q3_BudgetAllocationTracer": {"D": "D1", "Q": "Q3", "label": "Trazabilidad de Asignación Presupuestal", "methods": 13, "epistemic": ["structural", "financial", "normative"]},
    "D1_Q4_InstitutionalCapacityIdentifier": {"D": "D1", "Q": "Q4", "label": "Identificación de Capacidad Institucional", "methods": 11, "epistemic": ["semantic", "structural"]},
    "D1_Q5_ScopeJustificationValidator": {"D": "D1", "Q": "Q5", "label": "Validación de Justificación de Alcance", "methods": 7, "epistemic": ["temporal", "consistency", "normative"]},
    "D2_Q1_StructuredPlanningValidator": {"D": "D2", "Q": "Q1", "label": "Validación de Planificación Estructurada", "methods": 7, "epistemic": ["structural", "normative"]},
    "D2_Q2_InterventionLogicInferencer": {"D": "D2", "Q": "Q2", "label": "Inferencia de Lógica de Intervención", "methods": 11, "epistemic": ["causal", "bayesian", "structural"]},
    "D2_Q3_RootCauseLinkageAnalyzer": {"D": "D2", "Q": "Q3", "label": "Análisis de Vinculación a Causas Raíz", "methods": 9, "epistemic": ["causal", "structural", "semantic"]},
    "D2_Q4_RiskManagementAnalyzer": {"D": "D2", "Q": "Q4", "label": "Análisis de Gestión de Riesgos", "methods": 10, "epistemic": ["bayesian", "statistical", "normative"]},
    "D2_Q5_StrategicCoherenceEvaluator": {"D": "D2", "Q": "Q5", "label": "Evaluación de Coherencia Estratégica", "methods": 8, "epistemic": ["consistency", "normative", "structural"]},
    "D3_Q1_IndicatorQualityValidator": {"D": "D3", "Q": "Q1", "label": "Validación de Calidad de Indicadores", "methods": 8, "epistemic": ["normative", "structural", "semantic"]},
    "D3_Q3_TraceabilityValidator": {"D": "D3", "Q": "Q3", "label": "DIM03_Q03_TRACEABILITY_BUDGET_ORG", "methods": 22, "epistemic": ["structural", "semantic", "normative"]},
    "D3_Q4_TechnicalFeasibilityEvaluator": {"D": "D3", "Q": "Q4", "label": "DIM03_Q04_TECHNICAL_FEASIBILITY", "methods": 10, "epistemic": ["financial", "normative", "statistical"]},
    "D3_Q5_OutputOutcomeLinkageAnalyzer": {"D": "D3", "Q": "Q5", "label": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE", "methods": 9, "epistemic": ["causal", "structural", "semantic"]},
    "D4_Q1_OutcomeMetricsValidator": {"D": "D4", "Q": "Q1", "label": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS", "methods": 8, "epistemic": ["normative", "statistical", "semantic"]},
    "D4_Q2_CausalChainValidator": {"D": "D4", "Q": "Q2", "label": "Validación de Cadena Causal", "methods": 10, "epistemic": ["causal", "structural", "consistency"]},
    "D4_Q3_AmbitionJustificationAnalyzer": {"D": "D4", "Q": "Q3", "label": "Análisis de Justificación de Ambición", "methods": 9, "epistemic": ["normative", "statistical",

```
"semantic"]},
     "D4_Q4_ProblemSolvencyEvaluator": {"D": "D4", "Q": "Q4", "label": "Evaluación de Solvencia del Problema", "methods": 11, "epistemic": ["causal", "bayesian", "normative"]},
     "D4_Q5_VerticalAlignmentValidator": {"D": "D4", "Q": "Q5", "label": "Validación de Alineación Vertical", "methods": 8, "epistemic": ["structural", "consistency", "normative"]},
     "D5_Q1_LongTermVisionAnalyzer": {"D": "D5", "Q": "Q1", "label": "Análisis de Visión de Largo Plazo", "methods": 8, "epistemic": ["semantic", "normative", "temporal"]},
          "D5_Q2_CompositeMeasurementValidator": {"D": "D5", "Q": "Q2", "label": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY", "methods": 10, "epistemic": ["statistical", "normative", "semantic"]},
     "D5_Q3_IntangibleMeasurementAnalyzer": {"D": "D5", "Q": "Q3", "label": "Análisis de Medición Intangible", "methods": 9, "epistemic": ["semantic", "normative", "bayesian"]},
     "D5_Q4_SystemicRiskEvaluator": {"D": "D5", "Q": "Q4", "label": "Evaluación de Riesgo Sistémico", "methods": 12, "epistemic": ["bayesian", "causal", "normative"]},
     "D5_Q5_RealismAndSideEffectsAnalyzer": {"D": "D5", "Q": "Q5", "label": "Análisis de Realismo y Efectos Colaterales", "methods": 11, "epistemic": ["causal", "bayesian", "normative"]},
     "D6_Q2_LogicalProportionalityValidator": {"D": "D6", "Q": "Q2", "label": "Validación de Proporcionalidad Lógica", "methods": 10, "epistemic": ["structural", "consistency", "normative"]},
     "D6_Q3_ValidationTestingAnalyzer": {"D": "D6", "Q": "Q3", "label": "Análisis de Pruebas de Validación", "methods": 13, "epistemic": ["causal", "bayesian", "statistical"]},
     "D6_Q4_FeedbackLoopAnalyzer": {"D": "D6", "Q": "Q4", "label": "Análisis de Bucles de Retroalimentación", "methods": 9, "epistemic": ["causal", "structural", "temporal"]},
     "D6_Q5_ContextualAdaptabilityEvaluator": {"D": "D6", "Q": "Q5", "label": "Evaluación de Adaptabilidad Contextual", "methods": 10, "epistemic": ["semantic", "normative", "causal"]},
}

output_dir = Path(__file__).parent / "executor_configs"

for eid, spec in configs.items():
    mc = spec["methods"]
    timeout = 300 if mc <= 15 else (600 if mc <= 20 else 900)
     mem = 1024 if ("causal" in spec["epistemic"] or "bayesian" in spec["epistemic"]) else 512

    config = {
        "executor_id": eid,
        "dimension": spec["D"],
        "question": spec["Q"],
        "canonical_label": spec["label"],
        "role": "SCORE_Q",
        "required_layers": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
        "runtime_parameters": {
            "timeout_s": timeout,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": mem,
            "enable_profiling": True,
```

```
            "seed": 42
        },
        "thresholds": {
            "min_quality_score": 0.5,
            "min_evidence_confidence": 0.7,
            "max_runtime_ms": timeout * 1000
        },
        "epistemic_mix": spec["epistemic"],
        "contextual_params": {
            "expected_methods": mc,
            "critical_methods": [],
            "dimension_label": f"DIM0{spec['D'][1]}",
            "question_label": spec["label"]
        }
    }

    with open(output_dir / f"{eid}.json", "w") as f:
        json.dump(config, f, indent=2, ensure_ascii=False)

    print(f"Generated {eid}.json")

print(f"\\nCreated {len(configs)} config files in {output_dir}")
```

src/farfan_pipeline/phases/Phase_two/carver.py

```
"""
Doctoral-Carver Narrative Synthesizer v2.1 (SOTA Edition + Macro Synthesis)
================================================================================

Genera respuestas PhD-level con estilo minimalista Raymond Carver:
- Precisión quirúrgica en cada afirmación
- Sin adornos retóricos vacíos
- Cada palabra respaldada por evidencia
- Honestidad brutal sobre limitaciones
- Razonamiento causal explícito

Fundamentos Teóricos:
- Rhetorical Structure Theory (Mann & Thompson, 1988)
- Dempster-Shafer Evidence Theory (belief functions)
- Causal Inference Framework (Pearl, 2009)
- Argument Mining (Stab & Gurevych, 2017)
- Calibrated Uncertainty Quantification (Gneiting & Raftery, 2007)

Arquitectura:
1.ContractInterpreter: Extrae semántica profunda del contrato v3
2.EvidenceGraph: Construye grafo causal de evidencia
3.GapAnalyzer: Análisis multi-dimensional de vacíos
4.BayesianConfidence: Inferencia calibrada de confianza
5.DimensionTheory: Estrategias teóricamente fundamentadas por D1-D6
6.CarverRenderer: Prosa minimalista con máximo impacto
7.MacroSynthesizer: Agregación holística con análisis PA×DIM (v2.1)

Invariantes:
[INV-001] Toda afirmación debe tener ?1 evidencia citada
[INV-002] Gaps críticos siempre aparecen en respuesta
[INV-003] Confianza debe ser calibrada (no optimista)
[INV-004] Estilo Carver:  oraciones cortas, verbos activos, sin adverbios
[INV-005] Macro synthesis con divergencia PA×DIM explícita (v2.1)

Author: F.A. R.F.A.N Pipeline
Version: 2.1.0-SOTA-MACRO
"""

from __future__ import annotations

import math
import re
import statistics
from abc import ABC, abstractmethod
from collections import defaultdict
from dataclasses import dataclass, field
from enum import Enum
from typing import (
    Any,
    Dict,
    List,
    Optional,
```

```
    Tuple,
    TypeAlias,
)

# Readability and style checking libraries
try:
    import textstat  # Flesch-Kincaid and readability metrics
except ImportError:
    textstat = None  # type: ignore

try:
    from proselint.tools import lint as proselint_check  # Style and clarity checking
except ImportError:
    proselint_check = None  # type: ignore


# ============================================================================
# TYPE SYSTEM
# ============================================================================

Confidence:  TypeAlias = float  # [0.0, 1.0]
BeliefMass: TypeAlias = float  # Dempster-Shafer belief
PlausibilityMass: TypeAlias = float  # Dempster-Shafer plausibility


class Dimension(Enum):
    """Las 6 dimensiones causales del modelo lógico."""
    D1_INSUMOS = "DIM01"      # Inputs: recursos, datos, diagnóstico
    D2_ACTIVIDADES = "DIM02"  # Activities: acciones, instrumentos
    D3_PRODUCTOS = "DIM03"    # Outputs: entregables, metas
    D4_RESULTADOS = "DIM04"   # Outcomes: cambios inmediatos
    D5_IMPACTOS = "DIM05"     # Impacts: cambios largo plazo
    D6_CAUSALIDAD = "DIM06"   # Causality: lógica, M&E, adaptación


class EvidenceStrength(Enum):
    """Fuerza de evidencia según jerarquía epistemológica."""
    DEFINITIVE = "definitive"      # Dato oficial verificable
    STRONG = "strong"              # Múltiples fuentes concordantes
    MODERATE = "moderate"          # Fuente única confiable
    WEAK = "weak"                  # Inferido o parcial
    ABSENT = "absent"              # No encontrado


class GapSeverity(Enum):
    """Severidad de gaps con implicaciones para scoring."""
    CRITICAL = "critical"     # Bloquea evaluación positiva
    MAJOR = "major"           # Reduce score significativamente
    MINOR = "minor"           # Nota pero no bloquea
    COSMETIC = "cosmetic"     # Mejora deseable


class ArgumentRole(Enum):
    """Roles argumentativos (RST-inspired)."""
```

```
    CLAIM = "claim"           # Afirmación principal
    EVIDENCE = "evidence"     # Soporte factual
    WARRANT = "warrant"       # Justificación del vínculo
    QUALIFIER = "qualifier"   # Limitación/condición
    REBUTTAL = "rebuttal"     # Contraargumento reconocido
    BACKING = "backing"       # Soporte del warrant


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

@dataclass(frozen=True)
class ExpectedElement:
    """Elemento esperado con semántica completa."""
    type: str
    required: bool
    minimum:  int
    category: str  # 'quantitative', 'qualitative', 'relational'
    weight: float  # Importancia relativa [0, 1]

    @classmethod
    def from_contract(cls, elem: Dict[str, Any]) -> ExpectedElement:
        """Factory desde contrato."""
        elem_type = elem.get("type", "")

        # Inferir categoría desde tipo
        quantitative_types = {
            "indicadores_cuantitativos", "series_temporales_años",
            "monto_presupuestario", "meta_cuantificada", "linea_base",
            "porcentaje", "tasa", "indice"
        }
        relational_types = {
            "logica_causal_explicita", "ruta_transmision",
            "vinculo_causal", "dependencia_temporal"
        }

        if any(t in elem_type for t in quantitative_types):
            category = "quantitative"
        elif any(t in elem_type for t in relational_types):
            category = "relational"
        else:
            category = "qualitative"

        # Peso basado en required y minimum
        base_weight = 0.8 if elem.get("required", False) else 0.4
        min_val = elem.get("minimum", 0)
        weight = min(1.0, base_weight + (min_val * 0.05))

        return cls(
            type=elem_type,
            required=elem.get("required", False),
            minimum=elem.get("minimum", 1 if elem.get("required") else 0),
            category=category,
```

```
            weight=weight,
        )


@dataclass
class EvidenceItem:
    """Item de evidencia con metadatos ricos."""
    element_type: str
    value: Any
    confidence: float
    source_method: str
    document_location: Optional[str] = None

    # Computed properties
    strength: EvidenceStrength = EvidenceStrength.MODERATE
    is_quantitative: bool = False

    def __post_init__(self):
        """Compute derived properties."""
        # Determinar fuerza
        if self.confidence >= 0.9:
            self.strength = EvidenceStrength.STRONG
        elif self.confidence >= 0.7:
            self.strength = EvidenceStrength.MODERATE
        else:
            self.strength = EvidenceStrength.WEAK

        # Detectar si es cuantitativo
        if isinstance(self.value, (int, float)):
            self.is_quantitative = True
        elif isinstance(self.value, str):
            # Check for numeric patterns
            self.is_quantitative = bool(re.search(r'\d+[.,]?\d*\s*%?', self.value))


@dataclass
class EvidenceGap:
    """Gap con análisis causal de implicaciones."""
    element_type: str
    expected:  int
    found: int
    severity: GapSeverity
    implication: str  # Por qué importa este gap
    remediation: str  # Qué haría falta

    @property
    def deficit(self) -> int:
        return max(0, self.expected - self.found)

    @property
    def fulfillment_ratio(self) -> float:
        if self.expected == 0:
            return 1.0
        return min(1.0, self.found / self.expected)
```

```
@dataclass
class ArgumentUnit:
    """Unidad argumentativa con rol retórico."""
    role: ArgumentRole
    content: str
    evidence_refs: List[str]  # IDs de evidencia que soportan
    confidence: float

    def render(self) -> str:
        """Render según rol."""
        if self.role == ArgumentRole.CLAIM:
            return self.content
        elif self.role == ArgumentRole.EVIDENCE:
            return f"- {self.content}"
        elif self.role == ArgumentRole.QUALIFIER:
            return f"*{self.content}*"
        elif self.role == ArgumentRole.REBUTTAL:
            return f"Sin embargo: {self.content}"
        return self.content


@dataclass
class BayesianConfidenceResult:
    """Resultado de inferencia bayesiana de confianza."""
    point_estimate: float
    belief:  BeliefMass  # Grado de creencia
    plausibility: PlausibilityMass  # Límite superior de creencia
    uncertainty: float  # Ignorancia epistémica
    interval_95: Tuple[float, float]

    @property
    def is_calibrated(self) -> bool:
        """Check if interval is well-calibrated."""
        width = self.interval_95[1] - self.interval_95[0]
        return width >= 0.1  # No over-confident

    def to_label(self) -> str:
        """Human-readable label."""
        if self.point_estimate >= 0.85:
            return "ALTA"
        elif self.point_estimate >= 0.70:
            return "MEDIA-ALTA"
        elif self.point_estimate >= 0.50:
            return "MEDIA"
        elif self.point_estimate >= 0.30:
            return "BAJA"
        else:
            return "MUY BAJA"


@dataclass
class CarverAnswer:
```

```
    """Respuesta estructurada estilo Carver."""
    # Core components
    verdict: str  # Una oración.Directa.Sin escape.
    evidence_statements: List[str]  # Hechos.Verificables.
    gap_statements: List[str]  # Vacíos.Sin disculpas.

    # Confidence
    confidence_result: BayesianConfidenceResult
    confidence_statement: str

    # Metadata
    question_text: str
    dimension: Dimension
    method_note: str

    # Argumentative structure
    argument_units:  List[ArgumentUnit] = field(default_factory=list)

    # Trace
    synthesis_trace: Dict[str, Any] = field(default_factory=dict)


# ============================================================================
# CONTRACT INTERPRETER
# ============================================================================

class ContractInterpreter:
    """
    Extrae semántica profunda del contrato v3.

    No solo lee campos - interpreta intención.
    """

    # Mapeo de dimensiones a requisitos epistemológicos
    DIMENSION_REQUIREMENTS = {
        Dimension.D1_INSUMOS: {
            "primary_need": "datos cuantitativos verificables",
            "evidence_type": "quantitative",
            "minimum_sources": 2,
            "temporal_requirement": True,
        },
        Dimension.D2_ACTIVIDADES: {
            "primary_need": "especificidad operativa",
            "evidence_type": "qualitative",
            "minimum_sources": 1,
            "temporal_requirement":  False,
        },
        Dimension.D3_PRODUCTOS:  {
            "primary_need":  "proporcionalidad meta-problema",
            "evidence_type":  "mixed",
            "minimum_sources":  1,
            "temporal_requirement": True,
        },
        Dimension.D4_RESULTADOS:  {
```

```
        "primary_need":  "indicadores medibles",
        "evidence_type": "quantitative",
        "minimum_sources": 1,
        "temporal_requirement": True,
    },
    Dimension.D5_IMPACTOS:  {
        "primary_need":  "teoría de cambio",
        "evidence_type": "relational",
        "minimum_sources": 1,
        "temporal_requirement": True,
    },
    Dimension.D6_CAUSALIDAD: {
        "primary_need": "lógica causal explícita",
        "evidence_type": "relational",
        "minimum_sources": 1,
        "temporal_requirement": False,
    },
}

@classmethod
def extract_dimension(cls, contract: Dict) -> Dimension:
    """Extrae dimensión con fallback inteligente."""
    identity = contract.get("identity", {})
    dim_id = identity.get("dimension_id", "")

    # Try direct match
    for dim in Dimension:
        if dim.value == dim_id:
            return dim

    # Fallback:  infer from base_slot
    base_slot = identity.get("base_slot", "")
    if base_slot:
        try:
            dim_num = int(base_slot[1])  # "D1-Q1" -> 1
            return list(Dimension)[dim_num - 1]
        except (IndexError, ValueError):
            pass

    return Dimension.D1_INSUMOS  # Default

@classmethod
def extract_expected_elements(cls, contract: Dict) -> List[ExpectedElement]:
    """Extrae elementos con semántica enriquecida."""
    question_context = contract.get("question_context", {})
    raw_elements = question_context.get("expected_elements", [])

    return [ExpectedElement.from_contract(e) for e in raw_elements]

@classmethod
def extract_question_intent(cls, contract: Dict) -> Dict[str, Any]:
    """Extrae intención profunda de la pregunta."""
    question_context = contract.get("question_context", {})
    question_text = question_context.get("question_text", "")
```

```
        # Analizar tipo de pregunta
        q_lower = question_text.lower()

        if any(q in q_lower for q in ["¿cuánto", "¿cuántos", "qué porcentaje"]):
            question_type = "quantitative"
        elif any(q in q_lower for q in ["¿existe", "¿hay", "¿tiene", "¿incluye"]):
            question_type = "existence"
        elif any(q in q_lower for q in ["¿cómo", "¿de qué manera"]):
            question_type = "process"
        elif any(q in q_lower for q in ["¿por qué", "¿cuál es la razón"]):
            question_type = "causal"
        else:
            question_type = "descriptive"

        # Extraer tema principal (policy area hint)
        policy_area = contract.get("identity", {}).get("policy_area_id", "")

        return {
            "question_text": question_text,
            "question_type": question_type,
            "policy_area": policy_area,
            "requires_numeric":  question_type == "quantitative",
            "requires_causal_logic": question_type in ("causal", "process"),
        }

    @classmethod
    def get_dimension_theory(cls, dimension: Dimension) -> Dict[str, Any]:
        """Obtiene teoría epistemológica de la dimensión."""
        return cls.DIMENSION_REQUIREMENTS.get(dimension, {})

    @classmethod
    def extract_method_metadata(cls, contract: Dict) -> Dict[str, Any]:
        """Extrae metadata de métodos usados."""
        method_binding = contract.get("method_binding", {})

        return {
            "method_count": method_binding.get("method_count", 0),
            "orchestration_mode": method_binding.get("orchestration_mode", "unknown"),
            "methods": [
                m.get("method_name", "unknown")
                for m in method_binding.get("methods", [])
            ][: 5],  # Top 5
        }


# ============================================================================
# EVIDENCE ANALYZER
# ============================================================================

class EvidenceAnalyzer:
    """
    Análisis profundo de evidencia con construcción de grafo causal.
    """
```

```
    @staticmethod
    def extract_items(evidence: Dict[str, Any]) -> List[EvidenceItem]:
        """Extrae items de evidencia estructurados."""
        items = []

        for elem in evidence.get("elements", []):
            if isinstance(elem, dict):
                items.append(EvidenceItem(
                    element_type=elem.get("type", "unknown"),
                    value=elem.get("value", elem.get("description", "")),
                    confidence=float(elem.get("confidence", 0.5)),
                    source_method=elem.get("source_method", "unknown"),
                    document_location=elem.get("page", elem.get("location")),
                ))

        return items

    @staticmethod
    def count_by_type(items: List[EvidenceItem]) -> Dict[str, int]:
        """Cuenta items por tipo."""
        counts:  Dict[str, int] = defaultdict(int)
        for item in items:
            counts[item.element_type] += 1
        return dict(counts)

    @staticmethod
    def group_by_type(items: List[EvidenceItem]) -> Dict[str, List[EvidenceItem]]:
        """Agrupa items por tipo."""
        groups: Dict[str, List[EvidenceItem]] = defaultdict(list)
        for item in items:
            groups[item.element_type].append(item)
        return dict(groups)

    @staticmethod
    def analyze_strength_distribution(items: List[EvidenceItem]) -> Dict[str, int]:
        """Analiza distribución de fuerza de evidencia."""
        distribution:  Dict[str, int] = defaultdict(int)
        for item in items:
            distribution[item.strength.value] += 1
        return dict(distribution)

    @staticmethod
        def  find_corroborations(items:  List[EvidenceItem])  ->  List[Tuple[EvidenceItem,
EvidenceItem]]:
        """
        Encuentra pares de evidencia que se corroboran.

        Corroboración: mismo tipo, diferentes fuentes, valores consistentes.
        """
        corroborations = []
        groups = EvidenceAnalyzer.group_by_type(items)

        for elem_type, group_items in groups.items():
```

```
            if len(group_items) < 2:
                continue

            # Check pairs
            for i, item1 in enumerate(group_items):
                for item2 in group_items[i+1:]:
                    if item1.source_method != item2.source_method:
                        # Different sources = potential corroboration
                        corroborations.append((item1, item2))

        return corroborations

    @staticmethod
        def  find_contradictions(items:  List[EvidenceItem])  ->  List[Tuple[EvidenceItem,
EvidenceItem, str]]:
        """
        Encuentra contradicciones en evidencia.

        Returns: List of (item1, item2, explanation)
        """
        contradictions = []
        groups = EvidenceAnalyzer.group_by_type(items)

        for elem_type, group_items in groups.items():
            if len(group_items) < 2:
                continue

            # Check for numeric contradictions
            numeric_items = [i for i in group_items if i.is_quantitative]
            if len(numeric_items) >= 2:
                values = []
                for item in numeric_items:
                    try:
                        # Extract numeric value
                        val_str = str(item.value)
                        nums = re.findall(r'[\d.]+', val_str)
                        if nums:
                            values.append((item, float(nums[0])))
                    except (ValueError, TypeError, IndexError):
                        # Skip evidence items with non-numeric values
                        pass

                if len(values) >= 2:
                    # Check for significant divergence (>50% difference)
                    for i, (item1, val1) in enumerate(values):
                        for item2, val2 in values[i+1:]:
                            if val1 > 0 and abs(val1 - val2) / val1 > 0.5:
                                contradictions.append((
                                    item1, item2,
                                    f"Divergencia numérica: {val1} vs {val2}"
                                ))

        return contradictions
```

```
# ============================================================================
# GAP ANALYZER
# ============================================================================

class GapAnalyzer:
    """
    Análisis multi-dimensional de gaps con implicaciones causales.
    """

    # Implicaciones por tipo de elemento faltante
    GAP_IMPLICATIONS = {
        "fuentes_oficiales": (
            "Sin fuentes oficiales, la credibilidad del diagnóstico es cuestionable.",
            "Citar fuentes como DANE, Medicina Legal, ICBF."
        ),
        "indicadores_cuantitativos": (
            "Sin indicadores numéricos, no hay línea base medible.",
            "Incluir tasas, porcentajes o valores absolutos con fuente."
        ),
        "series_temporales_años": (
            "Sin series temporales, no se puede evaluar tendencia.",
            "Presentar datos de al menos 3 años consecutivos."
        ),
        "cobertura_territorial_especificada": (
            "Sin especificación territorial, el alcance es ambiguo.",
            "Definir si es municipal, departamental o por zonas."
        ),
        "logica_causal_explicita": (
            "Sin lógica causal, la teoría de cambio es invisible.",
            "Explicitar cadena:  insumo ? actividad ? producto ? resultado."
        ),
        "poblacion_objetivo_definida": (
            "Sin población objetivo, no hay focalización.",
            "Definir grupo beneficiario con características específicas."
        ),
        "instrumento_especificado": (
            "Sin instrumentos, las actividades son abstractas.",
            "Nombrar programas, proyectos o mecanismos concretos."
        ),
        "meta_cuantificada": (
            "Sin metas cuantificadas, no hay accountability.",
            "Establecer valores objetivo con plazo."
        ),
        "linea_base_resultado":  (
            "Sin línea base, no se puede medir avance.",
            "Documentar situación inicial con fecha."
        ),
        "impacto_definido": (
            "Sin impactos definidos, el propósito final es difuso.",
            "Describir cambios de largo plazo esperados."
        ),
        "sistema_monitoreo": (
            "Sin sistema de monitoreo, no hay seguimiento.",
```

```
        "Especificar indicadores, frecuencia y responsables."
    ),
}

@classmethod
def identify_gaps(
    cls,
    expected:  List[ExpectedElement],
    found_counts: Dict[str, int],
    dimension: Dimension,
) -> List[EvidenceGap]:
    """
    Identifica gaps con severidad calibrada por dimensión.
    """
    gaps = []
    dim_theory = ContractInterpreter.get_dimension_theory(dimension)

    for elem in expected:
        found = found_counts.get(elem.type, 0)

        if found >= elem.minimum:
            continue  # No gap

        # Determinar severidad
        severity = cls._compute_severity(elem, found, dim_theory)

        # Obtener implicación y remediación
        implication, remediation = cls.GAP_IMPLICATIONS.get(
            elem.type,
            (f"Falta {elem.type}.", f"Agregar {elem.type}.")
        )

        gaps.append(EvidenceGap(
            element_type=elem.type,
            expected=elem.minimum,
            found=found,
            severity=severity,
            implication=implication,
            remediation=remediation,
        ))

    # Sort by severity
    severity_order = {
        GapSeverity.CRITICAL: 0,
        GapSeverity.MAJOR: 1,
        GapSeverity.MINOR:  2,
        GapSeverity.COSMETIC: 3,
    }
    gaps.sort(key=lambda g: severity_order[g.severity])

    return gaps

@classmethod
def _compute_severity(
```

```
        cls,
        elem: ExpectedElement,
        found: int,
        dim_theory: Dict[str, Any],
    ) -> GapSeverity:
        """Computa severidad basada en contexto dimensional."""

        # Critical if required and completely missing
        if elem.required and found == 0:
            return GapSeverity.CRITICAL

        # Check if matches dimension's primary need
        evidence_type = dim_theory.get("evidence_type", "")

        # Critical if element type matches dimension's evidence type and missing
        if elem.category == evidence_type and found == 0:
            return GapSeverity.CRITICAL

        # Major if required but partial
        if elem.required and found < elem.minimum:
            return GapSeverity.MAJOR

        # Major if high weight and missing
        if elem.weight >= 0.7 and found == 0:
            return GapSeverity.MAJOR

        # Minor for optional but expected
        if elem.minimum > 0 and found < elem.minimum:
            return GapSeverity.MINOR

        return GapSeverity.COSMETIC


# ============================================================================
# BAYESIAN CONFIDENCE ENGINE
# ============================================================================

class BayesianConfidenceEngine:
    """
    Inferencia bayesiana de confianza con calibración.

    Usa Dempster-Shafer para manejar incertidumbre epistémica.
    """

    @staticmethod
    def compute(
        items: List[EvidenceItem],
        gaps: List[EvidenceGap],
        corroborations: List[Tuple[EvidenceItem, EvidenceItem]],
        contradictions: List[Tuple[EvidenceItem, EvidenceItem, str]],
    ) -> BayesianConfidenceResult:
        """
        Computa confianza calibrada usando Dempster-Shafer.
        """
```

```
        if not items:
            return BayesianConfidenceResult(
                point_estimate=0.0,
                belief=0.0,
                plausibility=0.3,
                uncertainty=1.0,
                interval_95=(0.0, 0.3),
            )

        # 1. Base:  average confidence of evidence
        confidences = [i.confidence for i in items]
        base_conf = statistics.mean(confidences)

        # 2. Boost for corroborations
        corroboration_boost = min(0.15, len(corroborations) * 0.05)

        # 3. Penalty for contradictions
        contradiction_penalty = min(0.25, len(contradictions) * 0.1)

        # 4. Penalty for gaps
        critical_gaps = sum(1 for g in gaps if g.severity == GapSeverity.CRITICAL)
        major_gaps = sum(1 for g in gaps if g.severity == GapSeverity.MAJOR)
        gap_penalty = min(0.4, critical_gaps * 0.15 + major_gaps * 0.05)

        # 5. Compute belief mass (lower bound of confidence)
           belief = max(0.0, base_conf + corroboration_boost - contradiction_penalty -
gap_penalty)
        belief = belief * (1 - 0.1 * critical_gaps)  # Further reduce for critical gaps

        # 6. Compute plausibility (upper bound)
        # Plausibility = 1 - belief in negation
        plausibility = min(1.0, belief + 0.2)  # Some uncertainty margin

        # 7. Epistemic uncertainty
        uncertainty = plausibility - belief

        # 8. Point estimate (expected value under ignorance)
        # Use Hurwicz criterion with pessimism weight
        pessimism_weight = 0.6  # Be conservative
             point_estimate = pessimism_weight * belief + (1 - pessimism_weight) *
plausibility

        # 9. Calibrated interval using Wilson score
        n = len(items)
        z = 1.96  # 95% CI

        p = point_estimate
        denominator = 1 + z**2 / n
        center = (p + z**2 / (2*n)) / denominator
        margin = z * math.sqrt((p * (1 - p) + z**2 / (4*n)) / n) / denominator

        lower = max(0.0, center - margin - gap_penalty)
        upper = min(1.0, center + margin)
```

```
        return BayesianConfidenceResult(
            point_estimate=round(point_estimate, 3),
            belief=round(belief, 3),
            plausibility=round(plausibility, 3),
            uncertainty=round(uncertainty, 3),
            interval_95=(round(lower, 3), round(upper, 3)),
        )


# ============================================================================
# DIMENSION-SPECIFIC STRATEGIES
# ============================================================================

class DimensionStrategy(ABC):
    """Base class for dimension-specific strategies."""

    @property
    @abstractmethod
    def dimension(self) -> Dimension:
        pass

    @abstractmethod
    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        """Prefix for verdict based on dimension theory."""
        pass

    @abstractmethod
    def key_requirement(self) -> str:
        """Key requirement for this dimension."""
        pass

    @abstractmethod
    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        """Dimension-specific confidence interpretation."""
        pass


class D1InsumosStrategy(DimensionStrategy):
    """D1: Insumos - Diagnóstico y datos cuantitativos."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D1_INSUMOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "El diagnóstico carece de fundamento cuantitativo."
        return "El diagnóstico tiene base cuantitativa."

    def key_requirement(self) -> str:
        return "Datos numéricos de fuentes oficiales."

    def interpret_confidence(self, conf:  BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
```

```
            return "Los datos son verificables."
        return "Faltan datos verificables."


class D2ActividadesStrategy(DimensionStrategy):
    """D2: Actividades - Especificidad operativa."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D2_ACTIVIDADES

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "Las actividades son vagas."
        return "Las actividades están especificadas."

    def key_requirement(self) -> str:
        return "Instrumento, población y lógica definidos."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La especificación es operativa."
        return "Falta especificidad operativa."


class D3ProductosStrategy(DimensionStrategy):
    """D3: Productos - Proporcionalidad y metas."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D3_PRODUCTOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "Los productos no son proporcionales al problema."
        return "Los productos son proporcionales."

    def key_requirement(self) -> str:
        return "Metas cuantificadas y proporcionales."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La proporcionalidad es clara."
        return "La proporcionalidad es cuestionable."


class D4ResultadosStrategy(DimensionStrategy):
    """D4: Resultados - Indicadores de outcome."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D4_RESULTADOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
```

```
        if has_critical_gaps:
            return "Los resultados no son medibles."
        return "Los resultados tienen indicadores."

    def key_requirement(self) -> str:
        return "Indicadores con línea base y meta."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "Los indicadores permiten seguimiento."
        return "El seguimiento no es posible."


class D5ImpactosStrategy(DimensionStrategy):
    """D5: Impactos - Cambios de largo plazo."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D5_IMPACTOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "El impacto de largo plazo no está definido."
        return "El impacto está conceptualizado."

    def key_requirement(self) -> str:
        return "Teoría de cambio con horizonte temporal."

    def interpret_confidence(self, conf:  BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La teoría de cambio es plausible."
        return "La teoría de cambio es débil."


class D6CausalidadStrategy(DimensionStrategy):
    """D6: Causalidad - M&E y adaptación."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D6_CAUSALIDAD

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "La lógica causal no es explícita."
        return "La cadena causal está documentada."

    def key_requirement(self) -> str:
        return "Sistema de M&E con ciclos de aprendizaje."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "El sistema permite adaptación."
        return "No hay mecanismo de corrección."
```

```
def get_dimension_strategy(dimension:  Dimension) -> DimensionStrategy:
    """Factory for dimension strategies."""
    strategies = {
        Dimension.D1_INSUMOS: D1InsumosStrategy(),
        Dimension.D2_ACTIVIDADES: D2ActividadesStrategy(),
        Dimension.D3_PRODUCTOS: D3ProductosStrategy(),
        Dimension.D4_RESULTADOS: D4ResultadosStrategy(),
        Dimension.D5_IMPACTOS: D5ImpactosStrategy(),
        Dimension.D6_CAUSALIDAD:  D6CausalidadStrategy(),
    }
    return strategies.get(dimension, D1InsumosStrategy())


# ============================================================================
# READABILITY & STYLE CHECKER (Flesch-Kincaid + Proselint)
# ============================================================================

@dataclass
class ReadabilityMetrics:
    """Métricas de legibilidad según Flesch-Kincaid y Proselint."""
    flesch_reading_ease: Optional[float] = None  # 0-100, higher is easier
    flesch_kincaid_grade: Optional[float] = None  # US grade level
    gunning_fog: Optional[float] = None  # Years of education needed
    avg_sentence_length: Optional[float] = None  # Words per sentence
    avg_word_length: Optional[float] = None  # Characters per word
    proselint_errors: List[Dict[str, Any]] = field(default_factory=list)  # Style issues
    proselint_score: Optional[float] = None  # 0-1, higher is better

    def passes_carver_standards(self) -> bool:
        """
        Verifica si el texto cumple estándares Carver:
        - Flesch Reading Ease >= 60 (standard readability)
        - Grade level <= 12 (accesible a público educado)
        - Sentence length <= 20 words (oraciones cortas)
        - Proselint score >= 0.9 (sin errores críticos)
        """
        if self.flesch_reading_ease and self.flesch_reading_ease < 60:
            return False
        if self.flesch_kincaid_grade and self.flesch_kincaid_grade > 12:
            return False
        if self.avg_sentence_length and self.avg_sentence_length > 20:
            return False
        if self.proselint_score and self.proselint_score < 0.9:
            return False
        return True


class ReadabilityChecker:
    """
    Aplica Flesch-Kincaid y Proselint para garantizar claridad Carver.

    Invariantes:
    - Flesch Reading Ease >= 60 (legible para público general)
```

```
- Grade Level <= 12 (no requiere posgrado)
- Oraciones <= 20 palabras promedio
- Sin errores Proselint críticos
"""

@staticmethod
def check_text(text: str) -> ReadabilityMetrics:
    """Analiza texto con Flesch-Kincaid y Proselint."""
    metrics = ReadabilityMetrics()

    # Flesch-Kincaid metrics (via textstat)
    if textstat:
        try:
            metrics.flesch_reading_ease = textstat.flesch_reading_ease(text)
            metrics.flesch_kincaid_grade = textstat.flesch_kincaid_grade(text)
            metrics.gunning_fog = textstat.gunning_fog(text)

            # Sentence and word stats
            sentences = textstat.sentence_count(text)
            words = textstat.lexicon_count(text, removepunct=True)
            if sentences > 0:
                metrics.avg_sentence_length = words / sentences

            chars = sum(len(word) for word in text.split())
            if words > 0:
                metrics.avg_word_length = chars / words

        except Exception as e:
            # Textstat can fail on very short or malformed text
            pass

    # Proselint style checking
    if proselint_check:
        try:
            errors = proselint_check(text)
            if errors:
                metrics.proselint_errors = errors
                # Score: 1.0 - (error_count / 100), capped at 0
                metrics.proselint_score = max(0.0, 1.0 - len(errors) / 100.0)
            else:
                metrics.proselint_score = 1.0
        except Exception as e:
            # Proselint can fail on malformed text
            metrics.proselint_score = 1.0  # Assume OK if check fails

    return metrics

@staticmethod
def enforce_carver_style(text: str) -> Tuple[str, ReadabilityMetrics]:
    """
    Analiza y opcionalmente ajusta texto para cumplir estándares Carver.

    Returns:
        (texto_ajustado, metrics)
```

```
        """
        metrics = ReadabilityChecker.check_text(text)

        # Si el texto ya cumple estándares, retornar sin modificar
        if metrics.passes_carver_standards():
            return text, metrics

        # Ajustes automáticos simples
        adjusted_text = text

        # Split long sentences (if avg > 20 words)
        if metrics.avg_sentence_length and metrics.avg_sentence_length > 20:
            # Replace comma-separated clauses with periods
            adjusted_text = re.sub(r',\s+([a-záéíóúñ])', r'. \1', adjusted_text)
            adjusted_text = re.sub(r'\s+y\s+([a-záéíóúñ])', r'. \1', adjusted_text)

        # Re-check after adjustments
        metrics = ReadabilityChecker.check_text(adjusted_text)

        return adjusted_text, metrics


# ============================================================================
# CARVER RENDERER
# ============================================================================

class CarverRenderer:
    """
    Renderiza prosa estilo Raymond Carver.

    Principios:
    - Oraciones cortas.Sujeto-verbo-objeto.
    - Verbos activos.Sin pasiva.
    - Sin adverbios.Sin adjetivos innecesarios.
    - Cada palabra cuenta.Si sobra, eliminar.
    - La verdad es suficiente.Sin adornos.
    """

    # Type mappings (technical ? plain Spanish)
    TYPE_LABELS = {
        "fuentes_oficiales": "fuentes oficiales",
        "indicadores_cuantitativos": "indicadores numéricos",
        "series_temporales_años": "series temporales",
        "cobertura_territorial_especificada": "cobertura territorial",
        "instrumento_especificado": "instrumentos",
        "poblacion_objetivo_definida": "población objetivo",
        "logica_causal_explicita": "lógica causal",
        "riesgos_identificados": "riesgos",
        "mitigacion_propuesta": "mitigación",
        "impacto_definido": "impactos",
        "rezago_temporal": "horizonte temporal",
        "ruta_transmision": "ruta de transmisión",
        "proporcionalidad_meta_problema": "proporcionalidad",
        "linea_base_resultado": "línea base",
```

```
    "meta_resultado": "metas",
    "meta_cuantificada": "metas cuantificadas",
    "metrica_outcome": "métricas",
    "sistema_monitoreo": "sistema de monitoreo",
    "ciclos_aprendizaje": "ciclos de aprendizaje",
    "mecanismos_correccion": "mecanismos de corrección",
    "analisis_contextual": "análisis contextual",
    "enfoque_diferencial": "enfoque diferencial",
}

@classmethod
def humanize(cls, elem_type: str) -> str:
    """Convert technical type to plain Spanish."""
    return cls.TYPE_LABELS.get(elem_type, elem_type.replace("_", " "))

@classmethod
def render_verdict(
    cls,
    strategy: DimensionStrategy,
    gaps: List[EvidenceGap],
    items: List[EvidenceItem],
) -> str:
    """
    Render verdict:  una oración.Sin escape.
    """
    critical_gaps = [g for g in gaps if g.severity == GapSeverity.CRITICAL]
    has_critical = len(critical_gaps) > 0

    prefix = strategy.verdict_prefix(has_critical)

    if not items:
        return f"{prefix} No hay evidencia."

    if has_critical:
        missing = [cls.humanize(g.element_type) for g in critical_gaps[: 2]]
        return f"{prefix} Falta: {', '.join(missing)}."

    return prefix

@classmethod
def render_evidence_statements(
    cls,
    items: List[EvidenceItem],
    found_counts: Dict[str, int],
) -> List[str]:
    """
    Render evidence as facts.Short.Verifiable.
    """
    statements = []

    # Total
    total = len(items)
    if total > 0:
        statements.append(f"{total} elementos de evidencia.")
```

```
        # Top types (max 3)
                    sorted_types  =  sorted(found_counts.items(),  key=lambda  x:  x[1],
reverse=True)[:3]
        for elem_type, count in sorted_types:
            label = cls.humanize(elem_type)
            statements.append(f"{count} {label}.")

        # Strength distribution
        strong = sum(1 for i in items if i.strength == EvidenceStrength.STRONG)
        if strong > 0:
            statements.append(f"{strong} elementos con alta confianza.")

        return statements

    @classmethod
    def render_gap_statements(
        cls,
        gaps: List[EvidenceGap],
    ) -> List[str]:
        """
        Render gaps.No excuses.Just facts.
        """
        statements = []

        for gap in gaps[: 4]:  # Max 4 gaps
            label = cls.humanize(gap.element_type)

            if gap.found == 0:
                statements.append(f"No hay {label}.")
            else:
                statements.append(f"{gap.found} {label}.  Se necesitan {gap.expected}.")

        return statements

    @classmethod
    def render_confidence_statement(
        cls,
        conf: BayesianConfidenceResult,
        strategy: DimensionStrategy,
    ) -> str:
        """
        Render confidence.Honest.Calibrated.
        """
        label = conf.to_label()
        pct = int(conf.point_estimate * 100)

        interpretation = strategy.interpret_confidence(conf)

        return f"Confianza {label} ({pct}%). {interpretation}"

    @classmethod
    def render_method_note(cls, method_meta: Dict[str, Any]) -> str:
        """
```

```
        Render method note.Brief.At the end.
        """
        count = method_meta.get("method_count", 0)
        return f"Análisis con {count} métodos."

    @classmethod
    def render_full_answer(cls, answer: CarverAnswer) -> str:
        """
        Render complete answer in Carver style with readability enforcement.
        """
        sections = []

        # Question context
        sections.append(f"**Pregunta**: {answer.question_text}\n")

        # Verdict (the core)
        sections.append(f"## Respuesta\n\n{answer.verdict}\n")

        # Evidence (facts only)
        if answer.evidence_statements:
            sections.append("## Evidencia\n")
            for stmt in answer.evidence_statements:
                sections.append(f"- {stmt}")

        # Gaps (if any)
        if answer.gap_statements:
            sections.append("\n## Vacíos\n")
            for stmt in answer.gap_statements:
                sections.append(f"- {stmt}")

        # Confidence
        sections.append(f"\n## Confianza\n\n{answer.confidence_statement}")

        # Method note (discrete)
        sections.append(f"\n---\n*{answer.method_note}*")

        # Join all sections
        full_text = "\n".join(sections)

        # Apply Flesch-Kincaid and Proselint readability checking
        adjusted_text, metrics = ReadabilityChecker.enforce_carver_style(full_text)

        # Add readability report if metrics available
        if metrics.flesch_reading_ease or metrics.proselint_score:
            readability_note = "\n\n---\n**Métricas de Legibilidad**:\n"
            if metrics.flesch_reading_ease:
                                    readability_note  +=  f"-  Flesch  Reading  Ease:
{metrics.flesch_reading_ease:.1f} "
                 readability_note += ("(Fácil)" if metrics.flesch_reading_ease >= 60 else
"(Difícil)")
                readability_note += "\n"
            if metrics.flesch_kincaid_grade:
                                        readability_note  +=  f"-  Nivel  Educativo:
{metrics.flesch_kincaid_grade:.1f} grado\n"
```

```
            if metrics.avg_sentence_length:
                                        readability_note  +=  f"-  Longitud  Promedio:
{metrics.avg_sentence_length:.1f} palabras/oración\n"
            if metrics.proselint_score is not None:
                                        readability_note  +=  f"-  Calidad  Proselint:
{metrics.proselint_score:.0%}"
                if metrics.proselint_errors:
                                readability_note  +=  f"  ({len(metrics.proselint_errors)}
sugerencias)"
                readability_note += "\n"

            # Only add note if text meets Carver standards
            if metrics.passes_carver_standards():
                        readability_note  +=  "\n?  Cumple  estándares  Carver  de  claridad  y
concisión."

            adjusted_text += readability_note

        return adjusted_text


# ============================================================================
# MAIN SYNTHESIZER
# ============================================================================

class DoctoralCarverSynthesizer:
    """
    Sintetizador Doctoral-Carver v2.0 SOTA.

    Combina rigor académico con prosa minimalista.
    Cada afirmación respaldada.Cada gap reconocido.
    Sin adornos.Sin excusas.Solo verdad.
    """

    def __init__(self):
        self.interpreter = ContractInterpreter()
        self.analyzer = EvidenceAnalyzer()
        self.gap_analyzer = GapAnalyzer()
        self.confidence_engine = BayesianConfidenceEngine()
        self.renderer = CarverRenderer()

    def synthesize(
        self,
        evidence: Dict[str, Any],
        contract:  Dict[str, Any],
    ) -> str:
        """
        Sintetiza respuesta doctoral-Carver.

        Args:
            evidence:  Evidencia ensamblada (dict con "elements", etc.)
            contract: Contrato v3 completo

        Returns:
```

```
            Respuesta en markdown, estilo Carver
        """
        # 1. Interpret contract
        dimension = self.interpreter.extract_dimension(contract)
        expected_elements = self.interpreter.extract_expected_elements(contract)
        question_intent = self.interpreter.extract_question_intent(contract)
        method_meta = self.interpreter.extract_method_metadata(contract)

        # 2. Get dimension strategy
        strategy = get_dimension_strategy(dimension)

        # 3. Analyze evidence
        items = self.analyzer.extract_items(evidence)
        found_counts = self.analyzer.count_by_type(items)
        corroborations = self.analyzer.find_corroborations(items)
        contradictions = self.analyzer.find_contradictions(items)

        # 4. Identify gaps
               gaps  =  self.gap_analyzer.identify_gaps(expected_elements,  found_counts,
dimension)

        # 5. Compute bayesian confidence
        confidence = self.confidence_engine.compute(
            items, gaps, corroborations, contradictions
        )

        # 6. Render components
        verdict = self.renderer.render_verdict(strategy, gaps, items)
        evidence_stmts = self.renderer.render_evidence_statements(items, found_counts)
        gap_stmts = self.renderer.render_gap_statements(gaps)
        conf_stmt = self.renderer.render_confidence_statement(confidence, strategy)
        method_note = self.renderer.render_method_note(method_meta)

        # 7. Compose answer
        answer = CarverAnswer(
            verdict=verdict,
            evidence_statements=evidence_stmts,
            gap_statements=gap_stmts,
            confidence_result=confidence,
            confidence_statement=conf_stmt,
            question_text=question_intent["question_text"],
            dimension=dimension,
            method_note=method_note,
            synthesis_trace={
                "dimension": dimension.value,
                "items_count": len(items),
                "gaps_count": len(gaps),
                           "critical_gaps":   sum(1  for  g  in  gaps  if  g.severity  ==
GapSeverity.CRITICAL),
                "corroborations":  len(corroborations),
                "contradictions": len(contradictions),
                "confidence": confidence.point_estimate,
            }
        )
```

```
        # 8. Render final output
        return self.renderer.render_full_answer(answer)

    def synthesize_structured(
        self,
        evidence: Dict[str, Any],
        contract: Dict[str, Any],
    ) -> CarverAnswer:
        """
        Returns structured CarverAnswer instead of string.

        Useful for further processing or integration.
        """
        # Same logic as synthesize but returns answer object
        dimension = self.interpreter.extract_dimension(contract)
        expected_elements = self.interpreter.extract_expected_elements(contract)
        question_intent = self.interpreter.extract_question_intent(contract)
        method_meta = self.interpreter.extract_method_metadata(contract)

        strategy = get_dimension_strategy(dimension)

        items = self.analyzer.extract_items(evidence)
        found_counts = self.analyzer.count_by_type(items)
        corroborations = self.analyzer.find_corroborations(items)
        contradictions = self.analyzer.find_contradictions(items)

                gaps  =  self.gap_analyzer.identify_gaps(expected_elements,  found_counts,
dimension)

        confidence = self.confidence_engine.compute(
            items, gaps, corroborations, contradictions
        )

        verdict = self.renderer.render_verdict(strategy, gaps, items)
        evidence_stmts = self.renderer.render_evidence_statements(items, found_counts)
        gap_stmts = self.renderer.render_gap_statements(gaps)
        conf_stmt = self.renderer.render_confidence_statement(confidence, strategy)
        method_note = self.renderer.render_method_note(method_meta)

        return CarverAnswer(
            verdict=verdict,
            evidence_statements=evidence_stmts,
            gap_statements=gap_stmts,
            confidence_result=confidence,
            confidence_statement=conf_stmt,
            question_text=question_intent["question_text"],
            dimension=dimension,
            method_note=method_note,
            synthesis_trace={
                "dimension": dimension.value,
                "items_count":  len(items),
                "gaps_count": len(gaps),
                            "critical_gaps":  sum(1  for  g  in  gaps  if  g.severity  ==
```

```
GapSeverity.CRITICAL),
                "corroborations": len(corroborations),
                "contradictions": len(contradictions),
                "confidence": confidence.point_estimate,
            }
        )

    def synthesize_macro(
        self,
        meso_results: List[Any],  # List[MesoQuestionResult]
        coverage_matrix: Optional[Dict[Tuple[str, str], float]] = None,
         macro_question_text: str = "¿El Plan de Desarrollo presenta una visión integral
y coherente?",
    ) -> Dict[str, Any]:
        """
        Sintetiza respuesta macro-level con análisis de divergencia PA×DIM.

        Agregación holística de múltiples meso-questions con:
        - Análisis de cobertura PA×DIM (10 policy areas × 6 dimensions)
        - Identificación de divergencias críticas
        - Cálculo de score holístico calibrado
        - Generación de hallazgos, fortalezas y debilidades

        Args:
            meso_results: Lista de resultados de meso-questions
            coverage_matrix: Matriz PA×DIM con scores {("PA01", "DIM01"): 0.85, ...}
            macro_question_text: Texto de la pregunta macro

        Returns:
            Dict con estructura de MacroQuestionResult:
                - score: Score holístico 0-1
                - scoring_level: Nivel (excelente/bueno/aceptable/insuficiente)
                - hallazgos: Lista de hallazgos globales
                - recomendaciones: Lista de recomendaciones priorizadas
                - fortalezas: Fortalezas identificadas
                - debilidades: Debilidades identificadas (gaps)
                - divergence_analysis: Análisis PA×DIM detallado
        """
        # 1. Analizar cobertura PA×DIM si está disponible
        divergence_analysis = {}
        if coverage_matrix:
            divergence_analysis = self._analyze_pa_dim_divergence(coverage_matrix)

        # 2. Agregar scores de meso-questions
            meso_scores = [m.get("score", 0.0) if isinstance(m, dict) else getattr(m,
"score", 0.0)
                           for m in meso_results]

        if not meso_scores:
            base_score = 0.0
        else:
            # Promedio ponderado con penalización por varianza alta
            base_score = statistics.mean(meso_scores)
            if len(meso_scores) > 1:
```

```
        variance = statistics.variance(meso_scores)
        # Penalizar inconsistencia (varianza alta)
        variance_penalty = min(0.15, variance * 0.3)
        base_score = max(0.0, base_score - variance_penalty)

# 3. Ajustar score con análisis de divergencia
if divergence_analysis:
    coverage_score = divergence_analysis.get("overall_coverage", 1.0)
    critical_gaps_count = divergence_analysis.get("critical_gaps_count", 0)

    # Penalizar gaps críticos en PA×DIM
    gap_penalty = min(0.25, critical_gaps_count * 0.05)

    # Score final como promedio ponderado
    final_score = (0.7 * base_score + 0.3 * coverage_score) - gap_penalty
else:
    final_score = base_score

final_score = max(0.0, min(1.0, final_score))

# 4. Determinar nivel de scoring
if final_score >= 0.85:
    scoring_level = "excelente"
elif final_score >= 0.70:
    scoring_level = "bueno"
elif final_score >= 0.55:
    scoring_level = "aceptable"
else:
    scoring_level = "insuficiente"

# 5. Generar hallazgos globales
hallazgos = self._generate_macro_hallazgos(
    meso_results, divergence_analysis, final_score
)

# 6. Generar fortalezas y debilidades
fortalezas, debilidades = self._identify_strengths_weaknesses(
    meso_results, divergence_analysis
)

# 7. Generar recomendaciones priorizadas
recomendaciones = self._generate_macro_recommendations(
    debilidades, divergence_analysis
)

# 8. Construir resultado macro
return {
    "score": round(final_score, 3),
    "scoring_level": scoring_level,
    "aggregation_method": "holistic_assessment",
    "meso_results": meso_results,
    "n_meso_evaluated": len(meso_results),
    "hallazgos": hallazgos,
    "recomendaciones": recomendaciones,
```

```
            "fortalezas": fortalezas,
            "debilidades": debilidades,
            "divergence_analysis": divergence_analysis,
            "metadata": {
                "question_text": macro_question_text,
                "synthesis_method": "doctoral_carver_macro_v2",
                "base_score": round(base_score, 3),
                "coverage_adjusted": coverage_matrix is not None,
            }
        }

    def _analyze_pa_dim_divergence(
        self,
        coverage_matrix: Dict[Tuple[str, str], float]
    ) -> Dict[str, Any]:
        """
        Analiza divergencia en matriz PA×DIM (10×6 = 60 células).

        Identifica:
        - Cobertura global (% de células con score >= threshold)
        - Gaps críticos (células con score < 0.5)
        - PAs y DIMs con baja cobertura
        - Patrones de divergencia
        """
        if not coverage_matrix:
            return {}

        # Definir umbrales
        THRESHOLD_ACCEPTABLE = 0.55
        THRESHOLD_CRITICAL = 0.50

        # 1. Análisis global
        all_scores = list(coverage_matrix.values())
        if not all_scores:
            return {"overall_coverage": 0.0, "critical_gaps_count": 0}

        overall_coverage = statistics.mean(all_scores)
        cells_above_threshold = sum(1 for s in all_scores if s >= THRESHOLD_ACCEPTABLE)
         coverage_percentage = cells_above_threshold / len(all_scores) if all_scores else
0.0

        # 2. Identificar gaps críticos
        critical_gaps = [
            (pa, dim, score)
            for (pa, dim), score in coverage_matrix.items()
            if score < THRESHOLD_CRITICAL
        ]

        # 3. Análisis por Policy Area
        policy_areas = set(pa for (pa, dim) in coverage_matrix.keys())
        pa_scores = {}
        for pa in policy_areas:
            pa_cells = [score for (p, d), score in coverage_matrix.items() if p == pa]
            pa_scores[pa] = statistics.mean(pa_cells) if pa_cells else 0.0
```

```
            low_coverage_pas = [pa for pa, score in pa_scores.items() if score < THRESHOLD_ACCEPTABLE]

        # 4. Análisis por Dimension
        dimensions = set(dim for (pa, dim) in coverage_matrix.keys())
        dim_scores = {}
        for dim in dimensions:
            dim_cells = [score for (p, d), score in coverage_matrix.items() if d == dim]
            dim_scores[dim] = statistics.mean(dim_cells) if dim_cells else 0.0

            low_coverage_dims = [dim for dim, score in dim_scores.items() if score < THRESHOLD_ACCEPTABLE]

        # 5. Identificar patrones de divergencia
        divergence_patterns = []

        if low_coverage_pas:
            divergence_patterns.append(
                f"Áreas de política con baja cobertura: {', '.join(low_coverage_pas)}"
            )

        if low_coverage_dims:
            dim_names = {
                "DIM01": "Insumos",
                "DIM02": "Actividades",
                "DIM03": "Productos",
                "DIM04": "Resultados",
                "DIM05": "Impactos",
                "DIM06": "Causalidad"
            }
            dim_labels = [dim_names.get(d, d) for d in low_coverage_dims]
            divergence_patterns.append(
                f"Dimensiones con baja cobertura: {', '.join(dim_labels)}"
            )

        if critical_gaps:
            # Agrupar por PA
            gaps_by_pa = defaultdict(int)
            for pa, dim, score in critical_gaps:
                gaps_by_pa[pa] += 1

                        top_gap_pas = sorted(gaps_by_pa.items(), key=lambda x: x[1], reverse=True)[:3]
            if top_gap_pas:
                pa_list = [f"{pa} ({count} gaps)" for pa, count in top_gap_pas]
                divergence_patterns.append(
                    f"PAs con más gaps críticos: {', '.join(pa_list)}"
                )

        return {
            "overall_coverage": round(overall_coverage, 3),
            "coverage_percentage": round(coverage_percentage, 3),
            "total_cells": len(coverage_matrix),
```

```
            "cells_above_threshold": cells_above_threshold,
            "critical_gaps_count": len(critical_gaps),
            "critical_gaps": critical_gaps[:5],  # Top 5 para no sobrecargar
            "low_coverage_pas": low_coverage_pas,
            "low_coverage_dims": low_coverage_dims,
            "pa_scores": {pa: round(score, 3) for pa, score in pa_scores.items()},
            "dim_scores": {dim: round(score, 3) for dim, score in dim_scores.items()},
            "divergence_patterns": divergence_patterns,
        }

    def _generate_macro_hallazgos(
        self,
        meso_results: List[Any],
        divergence_analysis: Dict[str, Any],
        final_score: float,
    ) -> List[str]:
        """Genera hallazgos globales del análisis macro."""
        hallazgos = []

        # 1. Hallazgo sobre score global
        if final_score >= 0.85:
            hallazgos.append(
                      "El plan presenta un nivel excelente de integración y coherencia
global."
            )
        elif final_score >= 0.70:
            hallazgos.append(
                "El plan muestra un nivel bueno de articulación entre dimensiones."
            )
        elif final_score >= 0.55:
            hallazgos.append(
                "El plan alcanza un nivel aceptable de coherencia, con áreas de mejora."
            )
        else:
            hallazgos.append(
                       "El plan presenta deficiencias significativas en integración y
coherencia."
            )

        # 2. Hallazgos de meso-questions
        if meso_results:
            high_scoring_mesos = [
                m for m in meso_results
                if (isinstance(m, dict) and m.get("score", 0) >= 0.80) or
                   (hasattr(m, "score") and m.score >= 0.80)
            ]
            low_scoring_mesos = [
                m for m in meso_results
                if (isinstance(m, dict) and m.get("score", 0) < 0.55) or
                   (hasattr(m, "score") and m.score < 0.55)
            ]

            if high_scoring_mesos:
                hallazgos.append(
```

```
                    f"{len(high_scoring_mesos)} de {len(meso_results)} clusters muestran
alto desempeño."
                )

            if low_scoring_mesos:
                hallazgos.append(
                    f"{len(low_scoring_mesos)} clusters requieren atención prioritaria."
                )

        # 3. Hallazgos de divergencia PA×DIM
        if divergence_analysis:
            coverage_pct = divergence_analysis.get("coverage_percentage", 0.0)
            critical_gaps = divergence_analysis.get("critical_gaps_count", 0)

            if coverage_pct >= 0.80:
                hallazgos.append(
                            f"Cobertura PA×DIM: {coverage_pct:.0%} de células con nivel
aceptable."
                )
            else:
                hallazgos.append(
                     f"Cobertura PA×DIM insuficiente: solo {coverage_pct:.0%} de células
aceptables."
                )

            if critical_gaps > 0:
                hallazgos.append(
                    f"{critical_gaps} células críticas identificadas en matriz PA×DIM."
                )

            # Agregar patrones de divergencia
            patterns = divergence_analysis.get("divergence_patterns", [])
            hallazgos.extend(patterns[:2])  # Top 2 patrones

        return hallazgos

    def _identify_strengths_weaknesses(
        self,
        meso_results: List[Any],
        divergence_analysis: Dict[str, Any],
    ) -> Tuple[List[str], List[str]]:
        """Identifica fortalezas y debilidades globales."""
        fortalezas = []
        debilidades = []

        # Analizar meso-questions
        if meso_results:
            scores = [
                m.get("score", 0.0) if isinstance(m, dict) else getattr(m, "score", 0.0)
                for m in meso_results
            ]

            if scores:
                avg_score = statistics.mean(scores)
```

```
        if avg_score >= 0.75:
            fortalezas.append(
                "Consistencia alta entre clusters temáticos."
            )

        if len(scores) > 1:
            variance = statistics.variance(scores)
            if variance < 0.05:
                fortalezas.append(
                    "Homogeneidad en calidad de implementación."
                )
            elif variance > 0.15:
                debilidades.append(
                    "Heterogeneidad significativa entre clusters."
                )

# Analizar divergencia PA×DIM
if divergence_analysis:
    overall_cov = divergence_analysis.get("overall_coverage", 0.0)

    if overall_cov >= 0.80:
        fortalezas.append(
            "Cobertura equilibrada de policy areas y dimensiones."
        )
    elif overall_cov < 0.60:
        debilidades.append(
            "Cobertura deficiente en matriz PA×DIM."
        )

    low_pas = divergence_analysis.get("low_coverage_pas", [])
    if low_pas:
        debilidades.append(
            f"Déficit en áreas: {', '.join(low_pas[:3])}."
        )

    low_dims = divergence_analysis.get("low_coverage_dims", [])
    if low_dims:
        dim_names = {
            "DIM01": "Insumos", "DIM02": "Actividades",
            "DIM03": "Productos", "DIM04": "Resultados",
            "DIM05": "Impactos", "DIM06": "Causalidad"
        }
        dim_labels = [dim_names.get(d, d) for d in low_dims[:2]]
        debilidades.append(
            f"Débil en dimensiones: {', '.join(dim_labels)}."
        )

# Asegurar al menos un elemento en cada lista
if not fortalezas:
    fortalezas.append("Plan documentado y estructurado.")

if not debilidades:
    debilidades.append("Oportunidades de fortalecimiento identificadas.")
```

```
        return fortalezas, debilidades

    def _generate_macro_recommendations(
        self,
        debilidades: List[str],
        divergence_analysis: Dict[str, Any],
    ) -> List[str]:
        """Genera recomendaciones priorizadas basadas en debilidades y gaps."""
        recomendaciones = []

        # 1. Recomendaciones basadas en divergencia PA×DIM
        if divergence_analysis:
            overall_cov = divergence_analysis.get("overall_coverage")
            coverage_pct = divergence_analysis.get("coverage_percentage")
            critical_gaps = divergence_analysis.get("critical_gaps_count", 0)

            if critical_gaps > 5:
                recomendaciones.append(
                       f"PRIORIDAD ALTA: Abordar {critical_gaps} gaps críticos en matriz
PA×DIM."
                )

                  # Even without "critical" gaps, low average coverage should trigger
actionable guidance.
            if isinstance(overall_cov, (int, float)) and overall_cov < 0.80:
                recomendaciones.append(
                      f"Incrementar cobertura promedio PA×DIM (actual={overall_cov:.2f})
mediante ajustes de diseño y trazabilidad."
                )

            if isinstance(overall_cov, (int, float)) and overall_cov < 0.85:
                recomendaciones.append(
                       "Reducir brechas internas: priorizar policy areas/dimensiones con
menor cobertura aunque no sean 'críticas'."
                )

            if isinstance(coverage_pct, (int, float)) and coverage_pct < 0.90:
                recomendaciones.append(
                                  f"Aumentar celdas sobre umbral aceptable (?0.55):
cobertura_actual={coverage_pct:.0%}."
                )

            low_pas = divergence_analysis.get("low_coverage_pas", [])
            if low_pas:
                recomendaciones.append(
                    f"Fortalecer policy areas: {', '.join(low_pas[:2])}."
                )

            low_dims = divergence_analysis.get("low_coverage_dims", [])
            if low_dims:
                dim_names = {
                    "DIM01": "Insumos", "DIM02": "Actividades",
                    "DIM03": "Productos", "DIM04": "Resultados",
```

```
                    "DIM05": "Impactos", "DIM06": "Causalidad"
                }
                dim_labels = [dim_names.get(d, d) for d in low_dims[:2]]
                recomendaciones.append(
                    f"Reforzar dimensiones: {', '.join(dim_labels)}."
                )

            # Recomendación específica por dimensión débil
            dim_scores = divergence_analysis.get("dim_scores", {})
            if dim_scores:
                weakest_dim = min(dim_scores.items(), key=lambda x: x[1])
                if weakest_dim[1] < 0.60:
                    dim_recs = {
                                "DIM01": "Mejorar diagnóstico con datos cuantitativos
verificables.",
                                "DIM02": "Especificar instrumentos y población objetivo de
intervenciones.",
                                "DIM03": "Cuantificar metas y establecer proporcionalidad con
problema.",
                                "DIM04": "Definir indicadores de resultado con línea base.",
                                "DIM05": "Explicitar teoría de cambio y horizonte temporal de
impactos.",
                                "DIM06": "Documentar lógica causal y sistema de monitoreo y
evaluación.",
                    }
                    rec = dim_recs.get(weakest_dim[0])
                    if rec:
                        recomendaciones.append(rec)

        # 2. Recomendaciones basadas en debilidades
         if any("heterogeneidad" in d.lower() or "inconsistencia" in d.lower() for d in
debilidades):
            recomendaciones.append(
                "Estandarizar metodología de formulación entre clusters."
            )

        # 3. Recomendación general de integración
        recomendaciones.append(
            "Reforzar articulación transversal entre policy areas y dimensiones."
        )

        return recomendaciones[:8]  # Máximo 8 recomendaciones


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

__all__ = [
    # Enums
    "Dimension",
    "EvidenceStrength",
    "GapSeverity",
    "ArgumentRole",
```

```
    # Data structures
    "ExpectedElement",
    "EvidenceItem",
    "EvidenceGap",
    "ArgumentUnit",
    "BayesianConfidenceResult",
    "CarverAnswer",

    # Components
    "ContractInterpreter",
    "EvidenceAnalyzer",
    "GapAnalyzer",
    "BayesianConfidenceEngine",
    "DimensionStrategy",
    "CarverRenderer",

    # Main class
    "DoctoralCarverSynthesizer",

    # Factory
    "get_dimension_strategy",
]
```

src/farfan_pipeline/phases/Phase_two/contract_validator_cqvr.py

```
from __future__ import annotations

import re
from dataclasses import dataclass, field
from enum import Enum
from typing import Any


class TriageDecision(Enum):
    PRODUCCION = "PRODUCCION"
    REFORMULAR = "REFORMULAR"
    PARCHEAR = "PARCHEAR"


@dataclass
class CQVRScore:
    tier1_score: float
    tier2_score: float
    tier3_score: float
    total_score: float
    tier1_max: float = 55.0
    tier2_max: float = 30.0
    tier3_max: float = 15.0
    total_max: float = 100.0
    component_scores: dict[str, float] = field(default_factory=dict)
    component_details: dict[str, dict[str, Any]] = field(default_factory=dict)

    @property
    def tier1_percentage(self) -> float:
        return (self.tier1_score / self.tier1_max) * 100 if self.tier1_max > 0 else 0

    @property
    def tier2_percentage(self) -> float:
        return (self.tier2_score / self.tier2_max) * 100 if self.tier2_max > 0 else 0

    @property
    def tier3_percentage(self) -> float:
        return (self.tier3_score / self.tier3_max) * 100 if self.tier3_max > 0 else 0

    @property
    def total_percentage(self) -> float:
        return (self.total_score / self.total_max) * 100 if self.total_max > 0 else 0


@dataclass
class ContractTriageDecision:
    decision: TriageDecision
    score: CQVRScore
    blockers: list[str]
    warnings: list[str]
    recommendations: list[dict[str, Any]]
    rationale: str
```

```
    def is_production_ready(self) -> bool:
        return self.decision == TriageDecision.PRODUCCION

    def requires_reformulation(self) -> bool:
        return self.decision == TriageDecision.REFORMULAR

    def can_be_patched(self) -> bool:
        return self.decision == TriageDecision.PARCHEAR


class CQVRValidator:
    TIER1_THRESHOLD = 35.0
    TIER1_PRODUCTION_THRESHOLD = 45.0
    TOTAL_PRODUCTION_THRESHOLD = 80.0

    def __init__(self) -> None:
        self.blockers: list[str] = []
        self.warnings: list[str] = []
        self.recommendations: list[dict[str, Any]] = []

    def validate_contract(self, contract: dict[str, Any]) -> ContractTriageDecision:
        self.blockers = []
        self.warnings = []
        self.recommendations = []

        tier1_score = self._evaluate_tier1(contract)
        tier2_score = self._evaluate_tier2(contract)
        tier3_score = self._evaluate_tier3(contract)

        score = CQVRScore(
            tier1_score=tier1_score,
            tier2_score=tier2_score,
            tier3_score=tier3_score,
            total_score=tier1_score + tier2_score + tier3_score
        )

        decision = self._make_triage_decision(score)
        rationale = self._generate_rationale(decision, score)

        return ContractTriageDecision(
            decision=decision,
            score=score,
            blockers=self.blockers.copy(),
            warnings=self.warnings.copy(),
            recommendations=self.recommendations.copy(),
            rationale=rationale
        )

    def _evaluate_tier1(self, contract: dict[str, Any]) -> float:
        a1_score = self.verify_identity_schema_coherence(contract)
        a2_score = self.verify_method_assembly_alignment(contract)
        a3_score = self.verify_signal_requirements(contract)
        a4_score = self.verify_output_schema(contract)
```

```
        return a1_score + a2_score + a3_score + a4_score

    def _evaluate_tier2(self, contract: dict[str, Any]) -> float:
        b1_score = self.verify_pattern_coverage(contract)
        b2_score = self.verify_method_specificity(contract)
        b3_score = self.verify_validation_rules(contract)

        return b1_score + b2_score + b3_score

    def _evaluate_tier3(self, contract: dict[str, Any]) -> float:
        c1_score = self.verify_documentation_quality(contract)
        c2_score = self.verify_human_template(contract)
        c3_score = self.verify_metadata_completeness(contract)

        return c1_score + c2_score + c3_score

    def verify_identity_schema_coherence(self, contract: dict[str, Any]) -> float:
        identity = contract.get("identity", {})
        output_schema = contract.get("output_contract", {}).get("schema", {})
        properties = output_schema.get("properties", {})

        score = 0.0
        max_score = 20.0

        fields_to_check = {
            "question_id": 5.0,
            "policy_area_id": 5.0,
            "dimension_id": 5.0,
            "question_global": 3.0,
            "base_slot": 2.0
        }

        for field, points in fields_to_check.items():
            identity_value = identity.get(field)
            schema_prop = properties.get(field, {})
            schema_value = schema_prop.get("const")

            if identity_value is not None and schema_value is not None:
                if identity_value == schema_value:
                    score += points
                else:
                    self.blockers.append(
                        f"A1: Identity-Schema mismatch for '{field}': "
                        f"identity={identity_value}, schema={schema_value}"
                    )
            else:
                if identity_value is None:
                    self.blockers.append(f"A1: Missing '{field}' in identity")
                if schema_value is None:
                            self.warnings.append(f"A1: Missing const for '{field}' in
output_schema")

        return score
```

```
def verify_method_assembly_alignment(self, contract: dict[str, Any]) -> float:
    method_binding = contract.get("method_binding", {})
    methods = method_binding.get("methods", [])
    evidence_assembly = contract.get("evidence_assembly", {})
    assembly_rules = evidence_assembly.get("assembly_rules", [])

    score = 0.0
    max_score = 20.0

    if not methods:
        self.blockers.append("A2: No methods defined in method_binding")
        return 0.0

    provides_set = set()
    for method in methods:
        provides = method.get("provides", "")
        if provides:
            provides_set.add(provides)

    method_count_declared = method_binding.get("method_count", len(methods))
    if method_count_declared == len(methods):
        score += 3.0
    else:
        self.warnings.append(
            f"A2: method_count mismatch: "
            f"declared={method_count_declared}, actual={len(methods)}"
        )

    sources_referenced = set()
    orphan_sources = []

    for rule in assembly_rules:
        sources = rule.get("sources", [])
        for source in sources:
            if isinstance(source, dict):
                source_key = source.get("namespace", "")
                if source_key and not source_key.startswith("*."):
                    sources_referenced.add(source_key)
                    if source_key not in provides_set:
                        orphan_sources.append(source_key)
            elif isinstance(source, str):
                if not source.startswith("*."):
                    sources_referenced.add(source)
                    if source not in provides_set:
                        orphan_sources.append(source)

    if not orphan_sources:
        score += 10.0
    else:
        self.blockers.append(
            f"A2: Assembly sources not in provides: {orphan_sources[:5]}"
        )
```

```
        usage_ratio = len(sources_referenced) / len(provides_set) if provides_set else 0
        if usage_ratio >= 0.9:
            score += 5.0
        elif usage_ratio >= 0.7:
            score += 3.0
        elif usage_ratio >= 0.5:
            score += 1.0
        else:
            self.warnings.append(
                f"A2: Low method usage ratio: {usage_ratio:.1%} "
                f"({len(sources_referenced)}/{len(provides_set)})"
            )

        if not orphan_sources:
            score += 2.0

        return score

    def verify_signal_requirements(self, contract: dict[str, Any]) -> float:
        signal_requirements = contract.get("signal_requirements", {})

        score = 0.0
        max_score = 10.0

        mandatory_signals = signal_requirements.get("mandatory_signals", [])
        threshold = signal_requirements.get("minimum_signal_threshold", 0.0)
        aggregation = signal_requirements.get("signal_aggregation", "")

        if mandatory_signals and threshold <= 0:
            self.blockers.append(
                f"A3: CRITICAL - minimum_signal_threshold={threshold} "
                "but mandatory_signals defined. "
                "This allows zero-strength signals to pass validation."
            )
            return 0.0

        if mandatory_signals and threshold > 0:
            score += 5.0
        elif not mandatory_signals:
            score += 5.0

        if mandatory_signals and all(isinstance(s, str) for s in mandatory_signals):
            score += 3.0
        elif mandatory_signals:
                    self.warnings.append("A3: Some mandatory_signals are not well-formed
strings")

        if aggregation in ["weighted_mean", "minimum", "product", "harmonic_mean"]:
            score += 2.0
        elif aggregation:
                        self.warnings.append(f"A3: Unknown signal_aggregation method:
{aggregation}")

        return score
```

```
def verify_output_schema(self, contract: dict[str, Any]) -> float:
    output_contract = contract.get("output_contract", {})
    schema = output_contract.get("schema", {})

    score = 0.0
    max_score = 5.0

    required = schema.get("required", [])
    properties = schema.get("properties", {})

    if not required:
        self.warnings.append("A4: No required fields in output_schema")
        return 0.0

    all_defined = all(field in properties for field in required)
    if all_defined:
        score += 3.0
    else:
        missing = [f for f in required if f not in properties]
        self.blockers.append(f"A4: Required fields not in properties: {missing}")

    traceability = contract.get("traceability", {})
    source_hash = traceability.get("source_hash", "")
    if source_hash and not source_hash.startswith("TODO"):
        score += 2.0
    else:
        self.warnings.append("A4: source_hash is placeholder or missing")
        score += 1.0

    return score

def verify_pattern_coverage(self, contract: dict[str, Any]) -> float:
    question_context = contract.get("question_context", {})
    patterns = question_context.get("patterns", [])
    expected_elements = question_context.get("expected_elements", [])

    score = 0.0
    max_score = 10.0

    if not expected_elements:
        self.warnings.append("B1: No expected_elements defined")
        return 0.0

    if not patterns:
        self.warnings.append("B1: No patterns defined")
        return 0.0

    required_elements = [e for e in expected_elements if e.get("required")]

    pattern_categories = set()
    for pattern in patterns:
        if isinstance(pattern, dict):
            category = pattern.get("category", "GENERAL")
```

```
            pattern_categories.add(category)

    coverage_score = min(len(patterns) / max(len(required_elements), 1) * 5.0, 5.0)
    score += coverage_score

    confidence_weights = [
        p.get("confidence_weight", 0) for p in patterns if isinstance(p, dict)
    ]
    if confidence_weights:
        valid_weights = all(0 <= w <= 1 for w in confidence_weights)
        if valid_weights:
            score += 3.0
        else:
            self.warnings.append("B1: Some confidence_weights out of [0,1] range")

    pattern_ids = [p.get("id", "") for p in patterns if isinstance(p, dict)]
    unique_ids = len(set(pattern_ids)) == len(pattern_ids)
    if unique_ids and all(pattern_ids):
        score += 2.0
    else:
        self.warnings.append("B1: Pattern IDs not unique or missing")

    return score

def verify_method_specificity(self, contract: dict[str, Any]) -> float:
    methodological_depth = contract.get("methodological_depth", {})
    methods = methodological_depth.get("methods", [])

    score = 0.0
    max_score = 10.0

    if not methods:
        self.warnings.append("B2: No methodological_depth.methods defined")
        return 0.0

    generic_patterns = [
        "Execute", "Process results", "Return structured output",
        "O(n) where n=input size", "Input data is preprocessed and valid"
    ]

    specific_count = 0
    boilerplate_count = 0

    for method_info in methods:
        technical = method_info.get("technical_approach", {})
        steps = technical.get("steps", [])
        complexity = technical.get("complexity", "")
        assumptions = technical.get("assumptions", [])

        is_specific = True

        for step in steps:
            step_desc = step.get("description", "")
            if any(pattern in step_desc for pattern in generic_patterns):
```

```
                    is_specific = False
                    boilerplate_count += 1
                    break

                if is_specific and complexity and not any(p in complexity for p in
generic_patterns):
                specific_count += 1

        if methods:
            specificity_ratio = specific_count / len(methods)
            score += specificity_ratio * 6.0

        complexity_count = sum(
            1 for m in methods
            if m.get("technical_approach", {}).get("complexity")
               and "input size" not in m.get("technical_approach", {}).get("complexity",
"")
        )
        if methods:
            score += (complexity_count / len(methods)) * 2.0

        assumptions_count = sum(
            1 for m in methods
            if m.get("technical_approach", {}).get("assumptions")
            and not any(
                "preprocessed" in str(a).lower()
                for a in m.get("technical_approach", {}).get("assumptions", [])
            )
        )
        if methods:
            score += (assumptions_count / len(methods)) * 2.0

        if boilerplate_count > len(methods) * 0.5:
            self.warnings.append(
                        f"B2: High boilerplate ratio: {boilerplate_count}/{len(methods)}
methods"
            )

        return score

    def verify_validation_rules(self, contract: dict[str, Any]) -> float:
        validation_rules = contract.get("validation_rules", {})
        rules = validation_rules.get("rules", [])
        question_context = contract.get("question_context", {})
        expected_elements = question_context.get("expected_elements", [])

        score = 0.0
        max_score = 10.0

        if not rules:
            self.blockers.append("B3: No validation_rules.rules defined")
            return 0.0

                required_elements = {e.get("type") for e in expected_elements if
```

```
e.get("required")}

        must_contain_elements = set()
        should_contain_elements = set()

        for rule in rules:
            must_contain = rule.get("must_contain", {})
            if isinstance(must_contain, dict):
                elements = must_contain.get("elements", [])
                must_contain_elements.update(elements)

            should_contain = rule.get("should_contain", [])
            if isinstance(should_contain, list):
                for item in should_contain:
                    if isinstance(item, dict):
                        elements = item.get("elements", [])
                        should_contain_elements.update(elements)

        all_validation_elements = must_contain_elements | should_contain_elements

        if required_elements and required_elements.issubset(all_validation_elements):
            score += 5.0
        elif required_elements:
            missing = required_elements - all_validation_elements
            self.warnings.append(
                f"B3: Required elements not in validation rules: {missing}"
            )

        if must_contain_elements and should_contain_elements:
            score += 3.0
        elif must_contain_elements or should_contain_elements:
            score += 1.0

        error_handling = contract.get("error_handling", {})
        failure_contract = error_handling.get("failure_contract", {})
        if failure_contract.get("emit_code"):
            score += 2.0
        else:
            self.warnings.append("B3: No emit_code in failure_contract")

        return score

    def verify_documentation_quality(self, contract: dict[str, Any]) -> float:
        methodological_depth = contract.get("methodological_depth", {})
        methods = methodological_depth.get("methods", [])

        score = 0.0
        max_score = 5.0

        if not methods:
            self.warnings.append("C1: No methodological_depth for documentation check")
            return 0.0

        boilerplate_patterns = [
```

```
            "analytical paradigm",
            "This method contributes",
            "method implements structured analysis"
        ]

        specific_paradigms = 0
        for method_info in methods:
            epist = method_info.get("epistemological_foundation", {})
            paradigm = epist.get("paradigm", "")
            justification = epist.get("justification", "")
            framework = epist.get("theoretical_framework", [])

            is_specific = True
            for pattern in boilerplate_patterns:
                        if pattern.lower() in paradigm.lower() or pattern.lower() in
justification.lower():
                    is_specific = False
                    break

            if is_specific:
                specific_paradigms += 1

        if methods:
            paradigm_ratio = specific_paradigms / len(methods)
            score += paradigm_ratio * 2.0

        justifications_with_why = sum(
            1 for m in methods
            if (
                   "why" in m.get("epistemological_foundation", {}).get("justification",
"").lower()
                 or "vs" in m.get("epistemological_foundation", {}).get("justification",
"").lower()
                or "alternative"
                        in m.get("epistemological_foundation", {}).get("justification",
"").lower()
            )
        )
        if methods:
            score += (justifications_with_why / len(methods)) * 2.0

        has_references = any(
            m.get("epistemological_foundation", {}).get("theoretical_framework")
            for m in methods
        )
        if has_references:
            score += 1.0

        return score

    def verify_human_template(self, contract: dict[str, Any]) -> float:
        output_contract = contract.get("output_contract", {})
        human_readable = output_contract.get("human_readable_output", {})
        template = human_readable.get("template", {})
```

```
        score = 0.0
        max_score = 5.0

        identity = contract.get("identity", {})
        base_slot = identity.get("base_slot", "")
        question_id = identity.get("question_id", "")

        title = template.get("title", "")
        summary = template.get("summary", "")

        has_references = False
        if base_slot and base_slot in title:
            has_references = True
        if question_id and question_id in title:
            has_references = True

        if has_references:
            score += 3.0
        else:
               self.warnings.append("C2: Template title does not reference base_slot or
question_id")

        placeholder_patterns = [
            r"\{.*?\}"
        ]

        has_placeholders = False
        for pattern in placeholder_patterns:
            if re.search(pattern, summary):
                has_placeholders = True
                break

        if has_placeholders:
            score += 2.0
        else:
            self.warnings.append("C2: Template summary has no dynamic placeholders")

        return score

    def verify_metadata_completeness(self, contract: dict[str, Any]) -> float:
        identity = contract.get("identity", {})
        traceability = contract.get("traceability", {})

        score = 0.0
        max_score = 5.0

        contract_hash = identity.get("contract_hash", "")
        if contract_hash and len(contract_hash) == 64:
            score += 2.0
        else:
            self.warnings.append("C3: contract_hash missing or invalid")

        created_at = identity.get("created_at", "")
```

```
        if created_at and "T" in created_at:
            score += 1.0
        else:
            self.warnings.append("C3: created_at missing or invalid ISO 8601 format")

        validated_against = identity.get("validated_against_schema", "")
        if validated_against:
            score += 1.0

        contract_version = identity.get("contract_version", "")
        if contract_version and "." in contract_version:
            score += 1.0

        source_hash = traceability.get("source_hash", "")
        if source_hash and not source_hash.startswith("TODO"):
            score += 3.0
        else:
                self.warnings.append("C3: source_hash is placeholder - breaks provenance
chain")
            self.recommendations.append({
                "component": "C3",
                "priority": "HIGH",
                "issue": "Missing source_hash",
                "fix": (
                    "Calculate SHA256 of questionnaire_monolith.json "
                    "and update traceability.source_hash"
                ),
                "impact": "+3 pts"
            })

        return min(score, max_score)

    def _make_triage_decision(self, score: CQVRScore) -> TriageDecision:
        if score.tier1_score < self.TIER1_THRESHOLD:
            return TriageDecision.REFORMULAR

        if (score.tier1_score >= self.TIER1_PRODUCTION_THRESHOLD and
            score.total_score >= self.TOTAL_PRODUCTION_THRESHOLD):
            return TriageDecision.PRODUCCION

        if len(self.blockers) == 0 and score.total_score >= 70:
            return TriageDecision.PARCHEAR

        if len(self.blockers) <= 2 and score.tier1_score >= 40:
            return TriageDecision.PARCHEAR

        return TriageDecision.REFORMULAR

    def _generate_rationale(self, decision: TriageDecision, score: CQVRScore) -> str:
        if decision == TriageDecision.PRODUCCION:
            return (
                f"Contract approved for production: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
```

```
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Blockers: {len(self.blockers)}, Warnings: {len(self.warnings)}."
            )
        elif decision == TriageDecision.PARCHEAR:
            return (
                f"Contract can be patched: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Blockers: {len(self.blockers)} (resolvable), "
                f"Warnings: {len(self.warnings)}. "
                f"Apply recommended patches to reach production threshold."
            )
        else:
            reason_parts = []
            if score.tier1_score < self.TIER1_THRESHOLD:
                            reason_parts.append(f"Tier 1 score below minimum threshold
({self.TIER1_THRESHOLD})")
            elif score.tier1_score < self.TIER1_PRODUCTION_THRESHOLD:
                        reason_parts.append(f"Tier 1 score below production threshold
({self.TIER1_PRODUCTION_THRESHOLD})")
            if score.total_score < self.TOTAL_PRODUCTION_THRESHOLD:
                         reason_parts.append(f"Total score below production threshold
({self.TOTAL_PRODUCTION_THRESHOLD})")
            if len(self.blockers) > 0:
                reason_parts.append(f"{len(self.blockers)} critical blocker(s)")

              reason_str = "; ".join(reason_parts) if reason_parts else "Failed decision
criteria"

            return (
                f"Contract requires reformulation: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Reasons: {reason_str}. "
                f"Contract needs substantial rework."
            )
```

src/farfan_pipeline/phases/Phase_two/evidence_nexus.py

```
"""
EvidenceNexus:  Unified SOTA Evidence-to-Answer Engine
=======================================================

REPLACES:
- evidence_assembler.py (merge strategies ? causal graph construction)
- evidence_validator.py (rule validation ? probabilistic consistency)
- evidence_registry.py (JSONL storage ? embedded vector store + hash chain)

ARCHITECTURE: Graph-Native Evidence Reasoning
---------------------------------------------
1.Evidence Ingestion ? Typed nodes in causal graph
2.Relationship Inference ? Edge weights via Bayesian inference
3.Consistency Validation ? Graph-theoretic conflict detection
4.Narrative Synthesis ? LLM-free template-driven answer generation
5.Provenance Tracking ? Merkle DAG with content-addressable storage

THEORETICAL FOUNDATIONS:
- Pearl's Causal Inference (do-calculus for counterfactual reasoning)
- Dempster-Shafer Theory (belief functions for uncertainty)
- Rhetorical Structure Theory (discourse coherence)
- Information-Theoretic Validation (mutual information for relevance)

INVARIANTS:
[INV-001] All evidence nodes must have SHA-256 content hash
[INV-002] Graph must be acyclic for causal reasoning
[INV-003] Narrative must cite ?1 evidence node per claim
[INV-004] Confidence intervals must be calibrated (coverage ? 0.95)
[INV-005] Hash chain must be append-only and verifiable

Author: F.A.R.F.A.N Pipeline
Version: 1.0.0
Date: 2025-12-10
"""

from __future__ import annotations

import hashlib
import json
import math
import re
import statistics
import time
from collections import defaultdict
from dataclasses import dataclass, field
from enum import Enum
from pathlib import Path
from typing import (
    Any,
    Protocol,
    Sequence,
    TypeAlias,
```

```
)

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


# ============================================================================
# TYPE SYSTEM
# ============================================================================

EvidenceID: TypeAlias = str  # SHA-256 hex digest
NodeID: TypeAlias = str
EdgeID: TypeAlias = str
Confidence: TypeAlias = float  # [0.0, 1.0]
BeliefMass: TypeAlias = float  # [0.0, 1.0]


class EvidenceType(Enum):
    """Taxonomy of evidence types aligned with questionnaire ontology."""
    # Quantitative
    INDICATOR_NUMERIC = "indicador_cuantitativo"
    TEMPORAL_SERIES = "serie_temporal"
    BUDGET_AMOUNT = "monto_presupuestario"
    COVERAGE_METRIC = "metrica_cobertura"
    GOAL_TARGET = "meta_cuantificada"

    # Qualitative
    OFFICIAL_SOURCE = "fuente_oficial"
    TERRITORIAL_COVERAGE = "cobertura_territorial"
    INSTITUTIONAL_ACTOR = "actor_institucional"
    POLICY_INSTRUMENT = "instrumento_politica"
    NORMATIVE_REFERENCE = "referencia_normativa"

    # Relational
    CAUSAL_LINK = "vinculo_causal"
    TEMPORAL_DEPENDENCY = "dependencia_temporal"
    CONTRADICTION = "contradiccion"
    CORROBORATION = "corroboracion"

    # Meta
    METHOD_OUTPUT = "salida_metodo"
    AGGREGATED = "agregado"
    SYNTHESIZED = "sintetizado"


class RelationType(Enum):
    """Edge types in evidence graph."""
    SUPPORTS = "supports"           # A provides evidence for B
    CONTRADICTS = "contradicts"     # A conflicts with B
    CAUSES = "causes"               # A is causal antecedent of B
```

```
    CORRELATES = "correlates"       # A and B co-occur without causation
    TEMPORALLY_PRECEDES = "precedes"  # A happens before B
    DERIVES_FROM = "derives"        # A was computed from B
    CITES = "cites"                 # A references B as source
    AGGREGATES = "aggregates"       # A is aggregation of B1... Bn


class ValidationSeverity(Enum):
    """Severity levels for validation findings."""
    CRITICAL = "critical"      # Blocks answer generation
    ERROR = "error"            # Degrades confidence significantly
    WARNING = "warning"        # Notes potential issue
    INFO = "info"              # Informational only


class AnswerCompleteness(Enum):
    """Classification of answer completeness."""
    COMPLETE = "complete"
    PARTIAL = "partial"
    INSUFFICIENT = "insufficient"
    NOT_APPLICABLE = "not_applicable"


class NarrativeSection(Enum):
    """Sections in structured narrative output."""
    DIRECT_ANSWER = "direct_answer"
    EVIDENCE_SUMMARY = "evidence_summary"
    CONFIDENCE_STATEMENT = "confidence_statement"
    SUPPORTING_DETAILS = "supporting_details"
    GAPS_AND_LIMITATIONS = "gaps_and_limitations"
    RECOMMENDATIONS = "recommendations"
    METHODOLOGY_NOTE = "methodology_note"


# ============================================================================
# CORE DATA STRUCTURES
# ============================================================================

@dataclass(frozen=True, slots=True)
class EvidenceNode:
    """
    Immutable evidence node with cryptographic identity.

    Each node represents a discrete piece of evidence extracted from
    document analysis.Nodes are content-addressed via SHA-256.

    Invariants:
    - node_id == SHA-256(canonical_json(content))
    - confidence in [0.0, 1.0]
    - belief_mass in [0.0, 1.0]
    - belief_mass + uncertainty_mass <= 1.0
    """
    node_id: EvidenceID
    evidence_type: EvidenceType
```

```
    content: dict[str, Any]

    # Confidence metrics
    confidence: Confidence
    belief_mass: BeliefMass  # Dempster-Shafer belief
    uncertainty_mass: float  # Epistemic uncertainty

    # Provenance
    source_method: str
    extraction_timestamp: float
    document_location: str | None  # Page/section reference

    # Metadata
    tags: frozenset[str] = field(default_factory=frozenset)
    parent_ids: tuple[EvidenceID, ... ] = field(default_factory=tuple)

    @classmethod
    def create(
        cls,
        evidence_type: EvidenceType,
        content: dict[str, Any],
        confidence: float,
        source_method: str,
        document_location: str | None = None,
        tags:  Sequence[str] | None = None,
        parent_ids:  Sequence[EvidenceID] | None = None,
        belief_mass: float | None = None,
        uncertainty_mass: float | None = None,
    ) -> EvidenceNode:
        """Factory method with automatic ID generation."""
        # Compute content hash for identity
        canonical = cls._canonical_json(content)
        node_id = hashlib.sha256(canonical.encode()).hexdigest()

        # Default belief mass to confidence if not specified
        bm = belief_mass if belief_mass is not None else confidence
         um = uncertainty_mass if uncertainty_mass is not None else (1.0 - confidence) *
0.5

        return cls(
            node_id=node_id,
            evidence_type=evidence_type,
            content=content,
            confidence=max(0.0, min(1.0, confidence)),
            belief_mass=max(0.0, min(1.0, bm)),
            uncertainty_mass=max(0.0, min(1.0, um)),
            source_method=source_method,
            extraction_timestamp=time.time(),
            document_location=document_location,
            tags=frozenset(tags or []),
            parent_ids=tuple(parent_ids or []),
        )

    @staticmethod
```

```
    def _canonical_json(obj: Any) -> str:
        """Deterministic JSON for hashing."""
        def default_handler(o:  Any) -> Any:
            if hasattr(o, '__dict__'):
                return o.__dict__
            if isinstance(o, Enum):
                return o.value
            return str(o)

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

    def to_dict(self) -> dict[str, Any]:
        """Serialize to dictionary."""
        return {
            "node_id": self.node_id,
            "evidence_type": self.evidence_type.value,
            "content": self.content,
            "confidence": self.confidence,
            "belief_mass": self.belief_mass,
            "uncertainty_mass": self.uncertainty_mass,
            "source_method": self.source_method,
            "extraction_timestamp": self.extraction_timestamp,
            "document_location": self.document_location,
            "tags": list(self.tags),
            "parent_ids": list(self.parent_ids),
        }


@dataclass(frozen=True, slots=True)
class EvidenceEdge:
    """
    Directed edge in evidence graph with Bayesian weight.

    Edges represent relationships between evidence nodes.
    Weight represents conditional probability P(target | source).
    """
    edge_id: EdgeID
    source_id: EvidenceID
    target_id: EvidenceID
    relation_type: RelationType
    weight:  float  # Conditional probability or strength
    confidence: Confidence
    metadata: dict[str, Any] = field(default_factory=dict)

    @classmethod
    def create(
        cls,
        source_id: EvidenceID,
        target_id: EvidenceID,
        relation_type: RelationType,
        weight: float = 1.0,
        confidence: float = 0.8,
        metadata: dict[str, Any] | None = None,
```

```
    ) -> EvidenceEdge:
        """Factory with auto-generated edge ID."""
        edge_id = hashlib.sha256(
            f"{source_id}:{target_id}:{relation_type.value}". encode()
        ).hexdigest()[:16]

        return cls(
            edge_id=edge_id,
            source_id=source_id,
            target_id=target_id,
            relation_type=relation_type,
            weight=max(0.0, min(1.0, weight)),
            confidence=max(0.0, min(1.0, confidence)),
            metadata=metadata or {},
        )


@dataclass
class ValidationFinding:
    """Single validation finding with severity and remediation."""
    finding_id: str
    severity: ValidationSeverity
    code: str
    message: str
    affected_nodes: list[EvidenceID]
    remediation: str | None = None

    def to_dict(self) -> dict[str, Any]:
        return {
            "finding_id":  self.finding_id,
            "severity": self.severity.value,
            "code": self.code,
            "message": self.message,
            "affected_nodes": self.affected_nodes,
            "remediation": self.remediation,
        }


@dataclass
class ValidationReport:
    """Complete validation report with aggregated findings."""
    is_valid: bool
    findings: list[ValidationFinding]
    critical_count: int
    error_count:  int
    warning_count: int
    validation_timestamp: float
    graph_integrity: bool
    consistency_score: float  # [0.0, 1.0]

    @classmethod
    def create(cls, findings: list[ValidationFinding]) -> ValidationReport:
        """Create report with computed aggregates."""
        critical = sum(1 for f in findings if f.severity == ValidationSeverity.CRITICAL)
```

```
        errors = sum(1 for f in findings if f.severity == ValidationSeverity.ERROR)
        warnings = sum(1 for f in findings if f.severity == ValidationSeverity.WARNING)

        # Valid only if no critical findings
        is_valid = critical == 0

        # Consistency score:  penalize errors and warnings
        base_score = 1.0
        base_score -= critical * 0.5  # Critical = -50% each
        base_score -= errors * 0.1    # Error = -10% each
        base_score -= warnings * 0.02 # Warning = -2% each
        consistency_score = max(0.0, base_score)

        return cls(
            is_valid=is_valid,
            findings=findings,
            critical_count=critical,
            error_count=errors,
            warning_count=warnings,
            validation_timestamp=time.time(),
            graph_integrity=True,  # Set by graph validation
            consistency_score=consistency_score,
        )


@dataclass
class Citation:
    """Evidence citation for narrative claims."""
    node_id: EvidenceID
    evidence_type: str
    value_summary: str
    confidence: float
    source_method: str
    document_reference: str | None = None

    def render(self, format_type: str = "markdown") -> str:
        """Render citation in specified format."""
        conf_pct = f"{self.confidence * 100:.0f}%"
        if format_type == "markdown":
            ref = f" (p.{self.document_reference})" if self.document_reference else ""
                return f"[{self.evidence_type}:  {self.value_summary}]{ref} (confianza:
{conf_pct})"
        return f"{self.evidence_type}: {self.value_summary} ({conf_pct})"


@dataclass
class NarrativeBlock:
    """Single block in structured narrative."""
    section: NarrativeSection
    content: str
    citations: list[Citation]
    confidence: float

    def render(self, format_type: str = "markdown") -> str:
```

```
        """Render block with citations."""
        if format_type == "markdown":
            header_map = {
                NarrativeSection.DIRECT_ANSWER:  "## Respuesta",
                NarrativeSection.EVIDENCE_SUMMARY: "### Resumen de Evidencia",
                NarrativeSection.CONFIDENCE_STATEMENT: "### Nivel de Confianza",
                NarrativeSection.SUPPORTING_DETAILS: "### Análisis Detallado",
                NarrativeSection.GAPS_AND_LIMITATIONS: "### Limitaciones Identificadas",
                NarrativeSection.RECOMMENDATIONS: "### Recomendaciones",
                NarrativeSection.METHODOLOGY_NOTE: "### Nota Metodológica",
            }
            header = header_map.get(self.section, f"### {self.section.value}")
            return f"{header}\n\n{self.content}"
        return f"{self.section.value.upper()}\n{self.content}"


@dataclass
class SynthesizedAnswer:
    """Complete synthesized answer with full provenance."""
    # Core answer
    direct_answer: str
    narrative_blocks: list[NarrativeBlock]

    # Quality metrics
    completeness: AnswerCompleteness
    overall_confidence: float
    calibrated_interval: tuple[float, float]  # 95% CI

    # Evidence linkage
    primary_citations: list[Citation]
    supporting_citations: list[Citation]

    # Gaps and issues
    gaps:  list[str]
    unresolved_contradictions: list[str]

    # Provenance
    evidence_graph_hash: str
    synthesis_timestamp: float
    question_id: str

    # Trace
    synthesis_trace: dict[str, Any]

    def to_human_readable(self, format_type: str = "markdown") -> str:
        """Generate final human-readable output."""
        sections = []

        for block in self.narrative_blocks:
            sections.append(block.render(format_type))

        separator = "\n\n" if format_type == "markdown" else "\n\n"
        return separator.join(sections)
```

```
    def to_dict(self) -> dict[str, Any]:
        """Full serialization."""
        return {
            "direct_answer": self.direct_answer,
            "completeness": self.completeness.value,
            "overall_confidence":  self.overall_confidence,
            "calibrated_interval": list(self.calibrated_interval),
            "gaps": self.gaps,
            "unresolved_contradictions": self.unresolved_contradictions,
            "evidence_graph_hash": self.evidence_graph_hash,
            "synthesis_timestamp":  self.synthesis_timestamp,
            "question_id": self.question_id,
            "primary_citation_count": len(self.primary_citations),
            "supporting_citation_count": len(self.supporting_citations),
        }


# ============================================================================
# EVIDENCE GRAPH
# ============================================================================

class EvidenceGraph:
    """
    Directed acyclic graph of evidence with causal reasoning support.

    Implements:
    - Content-addressable node storage
    - Relationship inference
    - Causal path analysis
    - Conflict detection
    - Belief propagation (Dempster-Shafer)
    """

    __slots__ = (
        '_nodes', '_edges', '_adjacency', '_reverse_adjacency',
        '_type_index', '_source_index', '_hash_chain', '_last_hash',
    )

    def __init__(self) -> None:
        self._nodes: dict[EvidenceID, EvidenceNode] = {}
        self._edges: dict[EdgeID, EvidenceEdge] = {}
        self._adjacency: dict[EvidenceID, list[EdgeID]] = defaultdict(list)
        self._reverse_adjacency: dict[EvidenceID, list[EdgeID]] = defaultdict(list)
        self._type_index: dict[EvidenceType, list[EvidenceID]] = defaultdict(list)
        self._source_index: dict[str, list[EvidenceID]] = defaultdict(list)
        self._hash_chain: list[str] = []
        self._last_hash: str | None = None

    # ------------------------------------------------------------------------
    # Node Operations
    # ------------------------------------------------------------------------

    def add_node(self, node:  EvidenceNode) -> EvidenceID:
        """Add node to graph with hash chain update."""
```

```
    if node.node_id in self._nodes:
        return node.node_id  # Idempotent

    self._nodes[node.node_id] = node
    self._type_index[node.evidence_type].append(node.node_id)
    self._source_index[node.source_method].append(node.node_id)

    # Update hash chain
    self._update_hash_chain(node)

    logger.debug("node_added", node_id=node.node_id[: 12],
                evidence_type=node.evidence_type.value)

    return node.node_id

def add_nodes(self, nodes:  Sequence[EvidenceNode]) -> list[EvidenceID]:
    """Batch add nodes."""
    return [self.add_node(n) for n in nodes]

def get_node(self, node_id:  EvidenceID) -> EvidenceNode | None:
    """Retrieve node by ID."""
    return self._nodes.get(node_id)

def get_nodes_by_type(self, evidence_type: EvidenceType) -> list[EvidenceNode]:
    """Get all nodes of a specific type."""
    node_ids = self._type_index.get(evidence_type, [])
    return [self._nodes[nid] for nid in node_ids if nid in self._nodes]

def get_nodes_by_source(self, source_method: str) -> list[EvidenceNode]:
    """Get all nodes from a specific source method."""
    node_ids = self._source_index.get(source_method, [])
    return [self._nodes[nid] for nid in node_ids if nid in self._nodes]

# ------------------------------------------------------------------------
# Edge Operations
# ------------------------------------------------------------------------

def add_edge(self, edge: EvidenceEdge) -> EdgeID:
    """Add edge to graph."""
    if edge.source_id not in self._nodes or edge.target_id not in self._nodes:
        raise ValueError(
            f"Cannot add edge:  source {edge.source_id[: 12]} or "
            f"target {edge.target_id[:12]} not in graph"
        )

    if edge.edge_id in self._edges:
        return edge.edge_id  # Idempotent

    # Check for cycle (DAG invariant)
    if self._would_create_cycle(edge.source_id, edge.target_id):
        raise ValueError(
            f"Cannot add edge: would create cycle from "
            f"{edge.source_id[:12]} to {edge.target_id[:12]}"
        )
```

```
        self._edges[edge.edge_id] = edge
        self._adjacency[edge.source_id].append(edge.edge_id)
        self._reverse_adjacency[edge.target_id].append(edge.edge_id)

        return edge.edge_id

    def _would_create_cycle(self, source:  EvidenceID, target: EvidenceID) -> bool:
        """Check if adding edge source?target would create a cycle."""
        # If target can reach source, adding source?target creates cycle
        visited:  set[EvidenceID] = set()
        stack = [target]

        while stack:
            current = stack.pop()
            if current == source:
                return True
            if current in visited:
                continue
            visited.add(current)

            # Follow outgoing edges
            for edge_id in self._adjacency.get(current, []):
                edge = self._edges[edge_id]
                stack.append(edge.target_id)

        return False

    def get_edges_from(self, node_id: EvidenceID) -> list[EvidenceEdge]:
        """Get outgoing edges from node."""
        edge_ids = self._adjacency.get(node_id, [])
        return [self._edges[eid] for eid in edge_ids if eid in self._edges]

    def get_edges_to(self, node_id: EvidenceID) -> list[EvidenceEdge]:
        """Get incoming edges to node."""
        edge_ids = self._reverse_adjacency.get(node_id, [])
        return [self._edges[eid] for eid in edge_ids if eid in self._edges]

    def get_edges_by_type(self, relation_type: RelationType) -> list[EvidenceEdge]:
        """Get all edges of a specific type."""
        return [e for e in self._edges.values() if e.relation_type == relation_type]

    # ------------------------------------------------------------------------
    # Graph Analysis
    # ------------------------------------------------------------------------

    def find_supporting_evidence(
        self,
        node_id: EvidenceID,
        max_depth: int = 3
    ) -> list[tuple[EvidenceNode, int]]:
        """
        Find all evidence that supports a node (transitive).
        Returns (node, depth) pairs.
```

```
        """
        results:  list[tuple[EvidenceNode, int]] = []
        visited:  set[EvidenceID] = set()

        def traverse(nid: EvidenceID, depth: int) -> None:
            if depth > max_depth or nid in visited:
                return
            visited.add(nid)

            for edge in self.get_edges_to(nid):
                                    if  edge.relation_type  in  (RelationType.SUPPORTS,
RelationType.DERIVES_FROM):
                    source_node = self._nodes.get(edge.source_id)
                    if source_node:
                        results.append((source_node, depth))
                        traverse(edge.source_id, depth + 1)

        traverse(node_id, 1)
        return results

           def  find_contradictions(self)  ->  list[tuple[EvidenceNode,  EvidenceNode,
EvidenceEdge]]:
        """Find all contradiction pairs in graph."""
        contradictions = []
        for edge in self.get_edges_by_type(RelationType.CONTRADICTS):
            source = self._nodes.get(edge.source_id)
            target = self._nodes.get(edge.target_id)
            if source and target:
                contradictions.append((source, target, edge))
        return contradictions

    def compute_belief_propagation(self) -> dict[EvidenceID, float]:
        """
        Dempster-Shafer belief propagation across graph.

        Combines evidence using Dempster's rule of combination.
        Returns updated belief masses for each node.
        """
        beliefs:  dict[EvidenceID, float] = {}

        # Topological sort for propagation order
        sorted_nodes = self._topological_sort()

        for node_id in sorted_nodes:
            node = self._nodes[node_id]
            incoming = self.get_edges_to(node_id)

            if not incoming:
                # Root node:  use intrinsic belief
                beliefs[node_id] = node.belief_mass
            else:
                # Combine beliefs from parents using Dempster's rule
                combined_belief = node.belief_mass
```

```
            for edge in incoming:
                if edge.relation_type == RelationType.SUPPORTS:
                    parent_belief = beliefs.get(edge.source_id, 0.5)
                    # Dempster's combination (simplified)
                    combined_belief = self._dempster_combine(
                        combined_belief, parent_belief * edge.weight
                    )
                elif edge.relation_type == RelationType.CONTRADICTS:
                    parent_belief = beliefs.get(edge.source_id, 0.5)
                    # Contradiction reduces belief
                    combined_belief *= (1 - parent_belief * edge.weight * 0.5)

            beliefs[node_id] = max(0.0, min(1.0, combined_belief))

    return beliefs

@staticmethod
def _dempster_combine(m1: float, m2: float) -> float:
    """Dempster's rule of combination for two belief masses."""
    # Simplified:  assume no direct conflict
    conflict = m1 * (1 - m2) * 0.1  # Small conflict factor
    normalization = 1 - conflict
    if normalization <= 0:
        return 0.5  # Maximum uncertainty

    combined = (m1 * m2) / normalization
    return max(0.0, min(1.0, combined))

def _topological_sort(self) -> list[EvidenceID]:
    """Topological sort of nodes (Kahn's algorithm)."""
    in_degree:  dict[EvidenceID, int] = {nid: 0 for nid in self._nodes}

    for edge in self._edges.values():
        in_degree[edge.target_id] += 1

    queue = [nid for nid, deg in in_degree.items() if deg == 0]
    result = []

    while queue:
        node_id = queue.pop(0)
        result.append(node_id)

        for edge in self.get_edges_from(node_id):
            in_degree[edge.target_id] -= 1
            if in_degree[edge.target_id] == 0:
                queue.append(edge.target_id)

    return result

# ------------------------------------------------------------------------
# Hash Chain (Provenance)
# ------------------------------------------------------------------------

def _update_hash_chain(self, node:  EvidenceNode) -> None:
```

```
        """Append node to hash chain."""
        chain_data = {
            "node_id": node.node_id,
            "previous_hash": self._last_hash or "",
            "timestamp": node.extraction_timestamp,
        }
        entry_hash = hashlib.sha256(
            json.dumps(chain_data, sort_keys=True).encode()
        ).hexdigest()

        self._hash_chain.append(entry_hash)
        self._last_hash = entry_hash

    def verify_hash_chain(self) -> bool:
        """Verify hash chain integrity."""
        if not self._hash_chain:
            return True

        # Would need full reconstruction to verify
        # For now, check chain exists and is non-empty
        return len(self._hash_chain) == len(self._nodes)

    def get_graph_hash(self) -> str:
        """Get hash of entire graph state."""
        if self._last_hash:
            return self._last_hash
        return hashlib.sha256(b"empty_graph").hexdigest()

    # ------------------------------------------------------------------------
    # Statistics
    # ------------------------------------------------------------------------

    @property
    def node_count(self) -> int:
        return len(self._nodes)

    @property
    def edge_count(self) -> int:
        return len(self._edges)

    def get_statistics(self) -> dict[str, Any]:
        """Comprehensive graph statistics."""
        type_counts = {t.value: len(ids) for t, ids in self._type_index.items()}
        source_counts = {s:  len(ids) for s, ids in self._source_index.items()}

        confidences = [n.confidence for n in self._nodes.values()]
        avg_confidence = statistics.mean(confidences) if confidences else 0.0

        edge_type_counts = defaultdict(int)
        for edge in self._edges.values():
            edge_type_counts[edge.relation_type.value] += 1

        return {
            "node_count": self.node_count,
```

```
            "edge_count": self.edge_count,
            "by_evidence_type": type_counts,
            "by_source_method": source_counts,
            "by_edge_type": dict(edge_type_counts),
            "average_confidence": avg_confidence,
            "hash_chain_length": len(self._hash_chain),
            "graph_hash": self.get_graph_hash()[: 16],
        }


# ============================================================================
# VALIDATION ENGINE
# ============================================================================

class ValidationRule(Protocol):
    """Protocol for validation rules."""

    @property
    def code(self) -> str: ...

    @property
    def severity(self) -> ValidationSeverity: ...

          def  validate(self,  graph:  EvidenceGraph,  contract:  dict[str,  Any])  ->
list[ValidationFinding]:  ...


class RequiredElementsRule:
    """Validate that required evidence types are present."""

    code = "REQ_ELEMENTS"
    severity = ValidationSeverity.ERROR

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

                                expected_elements    =    contract.get("question_context",
{}).get("expected_elements", [])

        for elem in expected_elements:
            elem_type = elem.get("type", "")
            required = elem.get("required", False)
            minimum = elem.get("minimum", 0)

            # In the monolith, expected_elements[].type includes many "context" gates
                                #   (e.g.,   coherencia_demostrada,   analisis_realismo,
trazabilidad_presupuestal)
            # that are NOT EvidenceType enum values. We must still validate them.
                     count,  node_ids  =  self._count_support_for_expected_element(graph,
str(elem_type))
```

```
            if required and count == 0:
                findings.append(ValidationFinding(
                    finding_id=f"REQ_{elem_type}",
                    severity=ValidationSeverity.ERROR,
                    code=self.code,
                            message=f"Required element type '{elem_type}' not found in
evidence",
                    affected_nodes=[],
                            remediation=f"Ensure document analysis extracts {elem_type}
elements",
                ))
            elif minimum > 0 and count < minimum:
                findings.append(ValidationFinding(
                    finding_id=f"MIN_{elem_type}",
                    severity=ValidationSeverity.WARNING,
                    code="MIN_ELEMENTS",
                      message=f"Element type '{elem_type}' has {count}/{minimum} required
instances",
                    affected_nodes=node_ids[:10],
                    remediation=f"Need {minimum - count} more {elem_type} elements",
                ))

        return findings

    @staticmethod
    def _count_support_for_expected_element(
        graph: EvidenceGraph,
        expected_type: str,
    ) -> tuple[int, list[str]]:
        """Count how much evidence supports an expected element type.

        Strategy:
        - Prefer exact EvidenceType matches when possible
                              -    Otherwise    use    contract-pattern    evidence    nodes
(source_method=contract.patterns)
          and map "context" types to categories/lexical markers.
        """
        expected = (expected_type or "").strip()
        if not expected:
            return 0, []

        # 1) Direct mapping to EvidenceType (when expected uses EvidenceType vocabulary)
        try:
            ev_type = EvidenceType(expected)
            nodes = graph.get_nodes_by_type(ev_type)
            return len(nodes), [n.node_id for n in nodes]
        except ValueError:
            pass

                  #  2)  Build  indices  from  contract.patterns  nodes  (produced  in
_build_graph_from_outputs)
        pattern_nodes = graph.get_nodes_by_source("contract.patterns")
        by_category: dict[str, list[EvidenceNode]] = defaultdict(list)
```

```
        for n in pattern_nodes:
            if isinstance(n.content, dict):
                cat = str(n.content.get("category") or "GENERAL").upper()
                by_category[cat].append(n)

                def _count_nodes_with_matches(nodes: list[EvidenceNode]) -> tuple[int,
list[str]]:
            ids: list[str] = []
            total = 0
            for n in nodes:
                mc = 0
                if isinstance(n.content, dict):
                    try:
                        mc = int(n.content.get("match_count", 0) or 0)
                    except Exception:
                        mc = 0
                if mc > 0:
                    total += mc
                    ids.append(n.node_id)
            return total, ids

        def _contains_any(match_texts: list[str], needles: tuple[str, ...]) -> bool:
            text = " ".join(match_texts).lower()
            return any(needle in text for needle in needles)

        # Helper to extract match texts from a node
        def _node_matches(node: EvidenceNode) -> list[str]:
            if not isinstance(node.content, dict):
                return []
            raw = node.content.get("matches", [])
            if isinstance(raw, list):
                return [str(x) for x in raw if x is not None]
            return []

        # 3) Map monolith expected element types ("contexts") to evidence signals
        # NOTE: This mapping is conservative and explainable. It can be refined
        # as we formalize context?pattern/validation specs in SISAS.
        et = expected.lower()

        # High-signal direct category mappings
        category_map: dict[str, str] = {
            "fuentes_oficiales": "FUENTE_OFICIAL",
            "indicadores_cuantitativos": "INDICADOR",
            "series_temporales_años": "TEMPORAL",
            "cobertura_territorial_especificada": "TERRITORIAL",
            "rezago_temporal": "TEMPORAL",
            "ruta_transmision": "CAUSAL",
            "logica_causal_explicita": "CAUSAL",
            "teoria_cambio_explicita": "CAUSAL",
            "cadena_causal_explicita": "CAUSAL",
            "mecanismo_causal_explicito": "CAUSAL",
            "unidades_medicion": "UNIDAD_MEDIDA",
            "trazabilidad_presupuestal": "INDICADOR",
        }
```

```
        if et in category_map:
            # CAUSAL is spread across CAUSAL*, so merge all keys starting with CAUSAL
            wanted = category_map[et]
            if wanted == "CAUSAL":
                causal_nodes: list[EvidenceNode] = []
                for k, nodes in by_category.items():
                    if k.startswith("CAUSAL"):
                        causal_nodes.extend(nodes)
                return _count_nodes_with_matches(causal_nodes)
            return _count_nodes_with_matches(by_category.get(wanted, []))

        # Context-style expected elements (from your list)
        if et == "completeness":
            # Treat as: any supporting evidence exists
            total_matches = 0
            ids: list[str] = []
            for nodes in by_category.values():
                c, nid = _count_nodes_with_matches(nodes)
                total_matches += c
                ids.extend(nid)
            return total_matches, ids

        if et == "horizonte_temporal":
            return _count_nodes_with_matches(by_category.get("TEMPORAL", []))

                  if  et  in  {"asignacion_explicita",  "restricciones_presupuestales",
"coherencia_recursos"}:
            # Budget/resource realism: indicator/unit + key lexemes in matched snippets.
            indicator_nodes = by_category.get("INDICADOR", [])
            unit_nodes = by_category.get("UNIDAD_MEDIDA", [])
            nodes = indicator_nodes + unit_nodes
            count, ids = _count_nodes_with_matches(nodes)
            if count == 0:
                return 0, []
            # Lexeme filter (keeps it honest)
             budget_lexemes = ("presupuesto", "recursos", "financi", "monto", "millones",
"cop", "$")
            supported_ids: list[str] = []
            supported_count = 0
            for n in nodes:
                matches = _node_matches(n)
                if matches and _contains_any(matches, budget_lexemes):
                    try:
                          supported_count += int(n.content.get("match_count", 0) or 0)  #
type: ignore[union-attr]
                    except Exception:
                        supported_count += 1
                    supported_ids.append(n.node_id)
            return (supported_count or count), (supported_ids or ids)

        if et in {"analisis_realismo", "analisis_contextual", "evidencia_comparada"}:
                                              #    Realism/context/comparison:    prefer
INDICADOR/TEMPORAL/FUENTE_OFICIAL/TERRITORIAL
```

```
            nodes = (
                by_category.get("INDICADOR", [])
                + by_category.get("TEMPORAL", [])
                + by_category.get("FUENTE_OFICIAL", [])
                + by_category.get("TERRITORIAL", [])
            )
            count, ids = _count_nodes_with_matches(nodes)
            if et == "evidencia_comparada" and count > 0:
                  compar_lex = ("compar", "vs", "promedio", "nacional", "departamental",
"anterior")
                supported_ids = []
                supported_count = 0
                for n in nodes:
                    matches = _node_matches(n)
                    if matches and _contains_any(matches, compar_lex):
                        supported_ids.append(n.node_id)
                        try:
                             supported_count += int(n.content.get("match_count", 0) or 0)
 # type: ignore[union-attr]
                        except Exception:
                            supported_count += 1
                return (supported_count or 0), supported_ids
            return count, ids

                     if   et   in   {"supuestos_identificados",   "riesgos_identificados",
"ciclos_aprendizaje",                "enfoque_diferencial",                "gobernanza",
"poblacion_objetivo_definida", "vinculo_diagnostico_actividad"}:
            # These are often GENERAL patterns with strong lexical anchors.
            general_nodes = by_category.get("GENERAL", [])
            count, ids = _count_nodes_with_matches(general_nodes)
            if count == 0:
                return 0, []
            anchors_by_type: dict[str, tuple[str, ...]] = {
                         "supuestos_identificados": ("supuesto", "asumi", "hipótesis",
"premisa"),
                "riesgos_identificados": ("riesgo", "amenaza", "mitig", "conting"),
                   "ciclos_aprendizaje": ("retroaliment", "aprendiz", "mejora", "ciclo",
"monitoreo"),
                    "enfoque_diferencial": ("enfoque diferencial", "enfoque de género",
"enfoque étn", "interseccional"),
                  "gobernanza": ("gobernanza", "coordinación", "articulación", "comité",
"mesa", "instancia"),
                   "poblacion_objetivo_definida": ("población objetivo", "beneficiari",
"grupo meta", "focaliz"),
                 "vinculo_diagnostico_actividad": ("diagnóstico", "brecha", "causa", "en
respuesta", "derivado"),
            }
            anchors = anchors_by_type.get(et, tuple())
            if not anchors:
                return count, ids
            supported_ids: list[str] = []
            supported_count = 0
            for n in general_nodes:
                matches = _node_matches(n)
```

```
                if matches and _contains_any(matches, anchors):
                    supported_ids.append(n.node_id)
                    try:
                         supported_count += int(n.content.get("match_count", 0) or 0)  #
type: ignore[union-attr]
                    except Exception:
                        supported_count += 1
            return supported_count, supported_ids

        # Fallback: treat as "any evidence exists" but only count nodes with matches
        total_matches = 0
        ids: list[str] = []
        for nodes in by_category.values():
            c, nid = _count_nodes_with_matches(nodes)
            total_matches += c
            ids.extend(nid)
        return total_matches, ids


class ConsistencyRule:
    """Validate internal consistency of evidence."""

    code = "CONSISTENCY"
    severity = ValidationSeverity.WARNING

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        # Check for unresolved contradictions
        contradictions = graph.find_contradictions()

        for source, target, edge in contradictions:
            if edge.confidence > 0.7:  # High-confidence contradiction
                findings.append(ValidationFinding(
                    finding_id=f"CONTRA_{edge.edge_id}",
                    severity=ValidationSeverity.WARNING,
                    code=self.code,
                    message=f"Contradiction detected between evidence nodes",
                    affected_nodes=[source.node_id, target.node_id],
                    remediation="Review contradictory evidence for resolution",
                ))

        return findings


class ConfidenceThresholdRule:
    """Validate confidence thresholds."""

    code = "CONFIDENCE"
    severity = ValidationSeverity.WARNING
```

```
    def __init__(self, min_confidence: float = 0.5):
        self.min_confidence = min_confidence

    def validate(
        self,
        graph: EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        low_confidence_nodes = [
            n for n in graph._nodes.values()
            if n.confidence < self.min_confidence
        ]

        if len(low_confidence_nodes) > graph.node_count * 0.3:
            findings.append(ValidationFinding(
                finding_id="LOW_CONF_AGGREGATE",
                severity=ValidationSeverity.WARNING,
                code=self.code,
                     message=f"{len(low_confidence_nodes)}/{graph.node_count} nodes have
confidence below {self.min_confidence}",
                affected_nodes=[n.node_id for n in low_confidence_nodes[: 10]],
                remediation="Consider additional evidence sources or validation",
            ))

        return findings


class GraphIntegrityRule:
    """Validate graph structural integrity."""

    code = "INTEGRITY"
    severity = ValidationSeverity.CRITICAL

    def validate(
        self,
        graph: EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        # Verify hash chain
        if not graph.verify_hash_chain():
            findings.append(ValidationFinding(
                finding_id="HASH_CHAIN_INVALID",
                severity=ValidationSeverity.CRITICAL,
                code=self.code,
                message="Hash chain integrity verification failed",
                affected_nodes=[],
                remediation="Evidence chain may be corrupted; rebuild from source",
            ))
```

```
        # Check for orphan edges
        for edge in graph._edges.values():
            if edge.source_id not in graph._nodes or edge.target_id not in graph._nodes:
                findings.append(ValidationFinding(
                    finding_id=f"ORPHAN_EDGE_{edge.edge_id}",
                    severity=ValidationSeverity.ERROR,
                    code=self.code,
                    message=f"Edge references non-existent node",
                    affected_nodes=[],
                    remediation="Remove orphan edge or add missing nodes",
                ))

        return findings


class ValidationEngine:
    """
    Probabilistic validation engine for evidence graphs.

    Replaces rule-based EvidenceValidator with graph-aware validation.
    """

    def __init__(self, rules: list[ValidationRule] | None = None):
        self.rules:  list[ValidationRule] = rules or [
            RequiredElementsRule(),
            ConsistencyRule(),
            ConfidenceThresholdRule(min_confidence=0.5),
            GraphIntegrityRule(),
        ]

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> ValidationReport:
        """Run all validation rules and produce report."""
        all_findings:  list[ValidationFinding] = []

        for rule in self.rules:
            try:
                findings = rule.validate(graph, contract)
                all_findings.extend(findings)
            except Exception as e:
                logger.error("validation_rule_failed", rule=rule.code, error=str(e))
                all_findings.append(ValidationFinding(
                    finding_id=f"RULE_ERROR_{rule.code}",
                    severity=ValidationSeverity.ERROR,
                    code="VALIDATION_ERROR",
                    message=f"Validation rule {rule.code} failed: {e}",
                    affected_nodes=[],
                ))

        report = ValidationReport.create(all_findings)
        report.graph_integrity = graph.verify_hash_chain()
```

```
        logger.info(
            "validation_complete",
            is_valid=report.is_valid,
            critical=report.critical_count,
            errors=report.error_count,
            warnings=report.warning_count,
        )

        return report


# ============================================================================
# NARRATIVE SYNTHESIZER
# ============================================================================

class NarrativeSynthesizer:
    """
    Transform evidence graph into coherent narrative answer.

    Implements Rhetorical Structure Theory for discourse coherence.
    """

    def __init__(
        self,
        citation_threshold: float = 0.6,
        max_citations_per_claim: int = 3,
    ):
        self.citation_threshold = citation_threshold
        self.max_citations_per_claim = max_citations_per_claim

    def synthesize(
        self,
        graph: EvidenceGraph,
        question_context: dict[str, Any],
        validation:  ValidationReport,
        contract: dict[str, Any],
    ) -> SynthesizedAnswer:
        """
        Synthesize complete answer from evidence graph.

        Process:
        1.Determine answer completeness from validation
        2.Select primary and supporting evidence
        3.Generate direct answer based on question type
        4.Build narrative blocks with citations
        5.Identify gaps and contradictions
        6.Compute calibrated confidence
        """
        question_global = question_context.get("question_global", "")
        question_id = question_context.get("question_id", "UNKNOWN")
        expected_elements = question_context.get("expected_elements", [])

        # 1. Determine completeness
```

```
                completeness = self._determine_completeness(graph, expected_elements,
validation)

        # 2. Select evidence
        primary_nodes = self._select_primary_evidence(graph, expected_elements)
        supporting_nodes = self._select_supporting_evidence(graph, primary_nodes)

        # 3. Build citations
        primary_citations = [self._node_to_citation(n) for n in primary_nodes]
        supporting_citations = [self._node_to_citation(n) for n in supporting_nodes]

        # 4. Generate direct answer
        answer_type = self._infer_answer_type(question_global)
        direct_answer = self._generate_direct_answer(
            graph, question_global, answer_type, completeness, primary_citations
        )

        # 5. Build narrative blocks
        blocks = self._build_narrative_blocks(
            direct_answer, graph, completeness, validation,
            primary_citations, supporting_citations
        )

        # 6. Identify gaps and contradictions
        gaps = self._identify_gaps(graph, expected_elements)
        contradictions = self._format_contradictions(graph.find_contradictions())

        # 7. Compute confidence
        overall_confidence, calibrated_interval = self._compute_confidence(
            graph, validation, completeness
        )

        return SynthesizedAnswer(
            direct_answer=direct_answer,
            narrative_blocks=blocks,
            completeness=completeness,
            overall_confidence=overall_confidence,
            calibrated_interval=calibrated_interval,
            primary_citations=primary_citations,
            supporting_citations=supporting_citations,
            gaps=gaps,
            unresolved_contradictions=contradictions,
            evidence_graph_hash=graph.get_graph_hash(),
            synthesis_timestamp=time.time(),
            question_id=question_id,
            synthesis_trace={
                "answer_type": answer_type,
                "primary_evidence_count": len(primary_nodes),
                "supporting_evidence_count":  len(supporting_nodes),
                "validation_passed": validation.is_valid,
            },
        )

    def _determine_completeness(
```

```
    self,
    graph: EvidenceGraph,
    expected_elements: list[dict[str, Any]],
    validation: ValidationReport,
) -> AnswerCompleteness:
    """Determine answer completeness based on evidence coverage."""
    if validation.critical_count > 0:
        return AnswerCompleteness.INSUFFICIENT

    required_types = [e["type"] for e in expected_elements if e.get("required")]
    found_types = set()

    for ev_type in EvidenceType:
        if graph.get_nodes_by_type(ev_type):
            found_types.add(ev_type.value)

    missing_required = set(required_types) - found_types

    if not missing_required:
        return AnswerCompleteness.COMPLETE
    elif len(found_types) > 0:
        return AnswerCompleteness.PARTIAL
    else:
        return AnswerCompleteness.INSUFFICIENT

def _select_primary_evidence(
    self,
    graph: EvidenceGraph,
    expected_elements: list[dict[str, Any]],
) -> list[EvidenceNode]:
    """Select primary evidence nodes for answer."""
    primary = []

    # Prioritize required elements
    required_types = [e["type"] for e in expected_elements if e.get("required")]

    for type_str in required_types:
        try:
            ev_type = EvidenceType(type_str)
            nodes = graph.get_nodes_by_type(ev_type)
            # Take highest confidence nodes
            sorted_nodes = sorted(nodes, key=lambda n: n.confidence, reverse=True)
            primary.extend(sorted_nodes[:self.max_citations_per_claim])
        except ValueError:
            continue

    # If no required types found, take highest confidence overall
    if not primary:
        all_nodes = list(graph._nodes.values())
        sorted_nodes = sorted(all_nodes, key=lambda n: n.confidence, reverse=True)
        primary = sorted_nodes[:5]

    return primary
```

```
def _select_supporting_evidence(
    self,
    graph: EvidenceGraph,
    primary_nodes: list[EvidenceNode],
) -> list[EvidenceNode]:
    """Select supporting evidence that corroborates primary."""
    supporting = []
    primary_ids = {n.node_id for n in primary_nodes}

    for node in primary_nodes:
        support = graph.find_supporting_evidence(node.node_id, max_depth=2)
        for supp_node, depth in support:
            if supp_node.node_id not in primary_ids:
                supporting.append(supp_node)

    # Deduplicate and limit
    seen = set()
    unique_supporting = []
    for n in supporting:
        if n.node_id not in seen:
            seen.add(n.node_id)
            unique_supporting.append(n)

    return unique_supporting[: 10]

def _node_to_citation(self, node: EvidenceNode) -> Citation:
    """Convert evidence node to citation."""
    value_summary = self._summarize_content(node.content)

    return Citation(
        node_id=node.node_id,
        evidence_type=node.evidence_type.value,
        value_summary=value_summary,
        confidence=node.confidence,
        source_method=node.source_method,
        document_reference=node.document_location,
    )

def _summarize_content(self, content:  dict[str, Any]) -> str:
    """Generate brief summary of evidence content."""
    # Try common fields
    for key in ["value", "text", "description", "name", "indicator"]:
        if key in content:
            val = content[key]
            if isinstance(val, str):
                return val[: 100] + ("..." if len(val) > 100 else "")
            return str(val)[:100]

    # Fallback:  first string value
    for val in content.values():
        if isinstance(val, str) and val:
            return val[:100]

    return str(content)[:100]
```

```
    def _infer_answer_type(self, question:  str) -> str:
        """Infer answer type from question text."""
        q_lower = question.lower()

         if any(q in q_lower for q in ["¿cuánto", "¿cuántos", "¿qué porcentaje", "¿cuál
es el monto"]):
            return "quantitative"
             if any(q in q_lower for q in ["¿existe", "¿hay", "¿tiene", "¿incluye",
"¿contempla"]):
            return "yes_no"
           if any(q in q_lower for q in ["¿cómo se compara", "¿cuál es mejor", "¿qué
diferencia"]):
            return "comparative"
        return "descriptive"

    def _generate_direct_answer(
        self,
        graph: EvidenceGraph,
        question:  str,
        answer_type:  str,
        completeness: AnswerCompleteness,
        citations: list[Citation],
    ) -> str:
        """Generate the direct answer to the question."""
        n_evidence = graph.node_count
        n_citations = len(citations)

        if completeness == AnswerCompleteness.INSUFFICIENT:
            return (
                f"**No se puede responder con confianza. ** El análisis del documento "
                f"no produjo suficiente evidencia para responder la pregunta: "
                 f"'{question[: 100]}... '.  Se identificaron solo {n_evidence} elementos
"
                      f"de evidencia, ninguno de los cuales cumple con los requisitos
mínimos."
            )

        if answer_type == "yes_no":
            if n_citations > 0:
                conf_avg = statistics.mean(c.confidence for c in citations)
                if conf_avg >= 0.7:
                    return (
                        f"**Sí**, el documento contiene evidencia positiva.  "
                        f"Se identificaron {n_citations} elementos que sustentan "
                                 f"una respuesta afirmativa con confianza promedio del
{conf_avg*100:.0f}%."
                    )
                else:
                    return (
                         f"**Parcialmente sí**, aunque con reservas.Se encontró evidencia
"
                              f"({n_citations} elementos), pero la confianza promedio es
{conf_avg*100:.0f}%, "
```

```
                    f"lo que sugiere información incompleta o ambigua."
                )
            return (
                 f"**No se encontró evidencia explícita** que responda afirmativamente.
"
                f"El documento analizado no contiene los elementos requeridos."
            )

        elif answer_type == "quantitative":
            # Look for numeric values in citations
            numeric_vals = []
            for c in citations:
                try:
                    # Try to extract number from value_summary
                    nums = re.findall(r'[\d,. ]+', c.value_summary)
                    if nums:
                        numeric_vals.append((c.evidence_type, nums[0]))
                except (AttributeError, TypeError):
                    # If value_summary is missing or not a string, skip this citation.
                    pass

            if numeric_vals:
                primary = numeric_vals[0]
                return (
                    f"El documento reporta **{primary[1]}** para {primary[0]}. "
                             f"Esta cifra se basa en {len(numeric_vals)} indicador(es)
cuantitativo(s) "
                    f"identificados en el análisis."
                )
            return (
                      f"El documento no especifica valores numéricos precisos para esta
pregunta.  "
                f"Se encontraron {n_citations} elementos de evidencia cualitativa que "
                f"pueden proporcionar contexto, pero no cifras exactas."
            )

        else:  # descriptive or comparative
            type_summary = ", ".join(set(c.evidence_type for c in citations[: 5]))

            quality = (
                "completa" if completeness == AnswerCompleteness.COMPLETE
                else "parcial"
            )

            return (
                f"El análisis del documento proporciona una respuesta **{quality}**. "
                   f"Se identificaron {n_evidence} elementos de evidencia en categorías
como: "
                f"{type_summary}. "
                 f"{'La información permite una evaluación confiable.' if completeness ==
AnswerCompleteness.COMPLETE else 'Se requiere información adicional para una evaluación
completa.'}"
            )
```

```
    def _build_narrative_blocks(
        self,
        direct_answer: str,
        graph: EvidenceGraph,
        completeness: AnswerCompleteness,
        validation: ValidationReport,
        primary_citations:  list[Citation],
        supporting_citations: list[Citation],
    ) -> list[NarrativeBlock]:
        """Build complete narrative structure."""
        blocks = []

        # 1. Direct Answer
        blocks.append(NarrativeBlock(
            section=NarrativeSection.DIRECT_ANSWER,
            content=direct_answer,
            citations=primary_citations[: 3],
            confidence=validation.consistency_score,
        ))

        # 2. Evidence Summary
        stats = graph.get_statistics()
        summary_content = (
            f"El análisis procesó evidencia de {stats['node_count']} elementos "
            f"con {stats['edge_count']} relaciones identificadas. "
            f"Confianza promedio: {stats['average_confidence']*100:.0f}%.  "
                                                    f"Distribución     por     tipo:
{self._format_type_distribution(stats['by_evidence_type'])}."
        )
        blocks.append(NarrativeBlock(
            section=NarrativeSection.EVIDENCE_SUMMARY,
            content=summary_content,
            citations=[],
            confidence=stats['average_confidence'],
        ))

        # 3. Confidence Statement
        conf_level = self._confidence_level_label(validation.consistency_score)
        conf_content = (
                                        f"**Nivel    de    confianza:       {conf_level}**
({validation.consistency_score*100:.0f}%). "
             f"Esta evaluación se basa en {len(primary_citations)} elementos de evidencia
primaria "
            f"y {len(supporting_citations)} elementos de soporte. "
              f"{'La validación no reportó errores críticos.' if validation.is_valid else
f'Se  identificaron  {validation.critical_count}  hallazgos  críticos  que  afectan  la
confianza.'}"
        )
        blocks.append(NarrativeBlock(
            section=NarrativeSection.CONFIDENCE_STATEMENT,
            content=conf_content,
            citations=[],
            confidence=validation.consistency_score,
        ))
```

```
        # 4. Supporting Details (if sufficient evidence)
        if primary_citations:
            details_parts = []
            for citation in primary_citations[: 5]:
                rendered = citation.render("markdown")
                details_parts.append(f"- {rendered}")

            details_content = (
                "**Evidencia principal identificada:**\n" +
                "\n".join(details_parts)
            )
            blocks.append(NarrativeBlock(
                section=NarrativeSection.SUPPORTING_DETAILS,
                content=details_content,
                citations=primary_citations[: 5],
                  confidence=statistics.mean(c.confidence for c in primary_citations) if
primary_citations else 0.0,
            ))

        return blocks

    def _identify_gaps(
        self,
        graph: EvidenceGraph,
        expected_elements: list[dict[str, Any]],
    ) -> list[str]:
        """Identify and describe evidence gaps."""
        gaps = []

        for elem in expected_elements:
            elem_type = elem.get("type", "")
            required = elem.get("required", False)
            minimum = elem.get("minimum", 0)

            try:
                ev_type = EvidenceType(elem_type)
                nodes = graph.get_nodes_by_type(ev_type)
                count = len(nodes)

                if required and count == 0:
                    gaps.append(
                        f"**{self._humanize_type(elem_type)}** (requerido): "
                        f"No se encontró evidencia de este tipo en el documento."
                    )
                elif minimum > 0 and count < minimum:
                    gaps.append(
                        f"**{self._humanize_type(elem_type)}**: "
                                f"Se encontraron {count} de {minimum} elementos mínimos
requeridos."
                    )
            except ValueError:
                continue
```

```
        # Check for low-confidence evidence clusters
        low_conf_types:  dict[str, int] = defaultdict(int)
        for node in graph._nodes.values():
            if node.confidence < 0.5:
                low_conf_types[node.evidence_type.value] += 1

        for ev_type, count in low_conf_types.items():
            if count >= 3:
                gaps.append(
                    f"**{self._humanize_type(ev_type)}**: "
                    f"{count} elementos tienen confianza baja (<50%), "
                    f"lo que sugiere extracción ambigua o fuentes poco claras."
                )

        return gaps

    def _format_contradictions(
        self,
        contradictions: list[tuple[EvidenceNode, EvidenceNode, EvidenceEdge]],
    ) -> list[str]:
        """Format contradictions for narrative."""
        formatted = []

        for source, target, edge in contradictions[: 5]:  # Limit to 5
            formatted.append(
                  f"Contradicción entre '{self._summarize_content(source.content)[: 50]}'
"
                f"y '{self._summarize_content(target.content)[:50]}' "
                f"(confianza del conflicto: {edge.confidence*100:.0f}%)"
            )

        return formatted

    def _compute_confidence(
        self,
        graph:  EvidenceGraph,
        validation: ValidationReport,
        completeness: AnswerCompleteness,
    ) -> tuple[float, tuple[float, float]]:
        """
        Compute overall confidence with calibrated interval.

        Returns (point_estimate, (lower_95, upper_95))
        """
        # Base confidence from validation
        base = validation.consistency_score

        # Adjust for completeness
        completeness_factor = {
            AnswerCompleteness.COMPLETE: 1.0,
            AnswerCompleteness.PARTIAL: 0.7,
            AnswerCompleteness.INSUFFICIENT: 0.3,
            AnswerCompleteness.NOT_APPLICABLE: 0.0,
        }[completeness]
```

```
        # Adjust for evidence quantity (diminishing returns)
        quantity_factor = min(1.0, math.log1p(graph.node_count) / math.log1p(50))

        # Combine factors
        point_estimate = base * completeness_factor * (0.5 + 0.5 * quantity_factor)
        point_estimate = max(0.0, min(1.0, point_estimate))

        # Calibrated interval (Wilson score interval approximation)
        n = max(1, graph.node_count)
        z = 1.96  # 95% CI

        denominator = 1 + z**2 / n
        center = (point_estimate + z**2 / (2*n)) / denominator
         margin = z * math.sqrt((point_estimate * (1 - point_estimate) + z**2 / (4*n)) /
n) / denominator

        lower = max(0.0, center - margin)
        upper = min(1.0, center + margin)

        return point_estimate, (lower, upper)

    def _format_type_distribution(self, type_counts: dict[str, int]) -> str:
        """Format type distribution for narrative."""
        if not type_counts:
            return "ninguno"

        sorted_types = sorted(type_counts.items(), key=lambda x: x[1], reverse=True)
        parts = [f"{self._humanize_type(t)}({c})" for t, c in sorted_types[: 4]]
        return ", ".join(parts)

    def _confidence_level_label(self, score: float) -> str:
        """Map confidence score to human label."""
        if score >= 0.85:
            return "ALTO"
        elif score >= 0.70:
            return "MEDIO-ALTO"
        elif score >= 0.50:
            return "MEDIO"
        elif score >= 0.30:
            return "BAJO"
        else:
            return "MUY BAJO"

    @staticmethod
    def _humanize_type(elem_type: str) -> str:
        """Convert element type to human-readable label."""
        mappings = {
            "indicador_cuantitativo": "indicadores cuantitativos",
            "serie_temporal": "series temporales",
            "monto_presupuestario": "montos presupuestarios",
            "metrica_cobertura": "métricas de cobertura",
            "meta_cuantificada": "metas cuantificadas",
            "fuente_oficial": "fuentes oficiales",
```

```
            "cobertura_territorial": "cobertura territorial",
            "actor_institucional": "actores institucionales",
            "instrumento_politica": "instrumentos de política",
            "referencia_normativa": "referencias normativas",
            "vinculo_causal": "vínculos causales",
            "dependencia_temporal": "dependencias temporales",
            "contradiccion": "contradicciones",
            "corroboracion": "corroboraciones",
            "fuentes_oficiales": "fuentes oficiales",
            "indicadores_cuantitativos": "indicadores cuantitativos",
            "series_temporales_años": "series temporales",
            "cobertura_territorial_especificada": "cobertura territorial",
        }
        return mappings.get(elem_type, elem_type.replace("_", " "))


# ============================================================================
# UNIFIED ENGINE:  EvidenceNexus
# ============================================================================

class EvidenceNexus:
    """
    Unified SOTA Evidence-to-Answer Engine.

    REPLACES:
    - EvidenceAssembler:  Graph-based evidence fusion
    - EvidenceValidator: Probabilistic graph validation
    - EvidenceRegistry: Embedded provenance with hash chain

    PROVIDES:
    - Causal graph construction from method outputs
    - Bayesian belief propagation
    - Conflict detection and resolution
    - Narrative synthesis with citations
    - Cryptographic provenance chain

    Usage:
        nexus = EvidenceNexus()
        result = nexus.process(
            method_outputs=method_outputs,
            question_context=question_context,
            contract=contract,
        )

        # Result contains:
        # - evidence_graph: Full graph with all nodes/edges
        # - validation_report: Comprehensive validation
        # - synthesized_answer: Complete narrative answer
        # - human_readable_output: Formatted string
    """

    def __init__(
        self,
        storage_path: Path | None = None,
```

```
    enable_persistence: bool = True,
    validation_rules: list[ValidationRule] | None = None,
    citation_threshold: float = 0.6,
):
    """
    Initialize EvidenceNexus.

    Args:
        storage_path: Path for persistent storage (JSONL)
        enable_persistence: Whether to persist to disk
        validation_rules:  Custom validation rules
        citation_threshold:  Minimum confidence for citation
    """
    self.storage_path = storage_path or Path("evidence_nexus.jsonl")
    self.enable_persistence = enable_persistence

    self.validation_engine = ValidationEngine(rules=validation_rules)
    self.narrative_synthesizer = NarrativeSynthesizer(
        citation_threshold=citation_threshold
    )

    # Current session graph
    self._graph:  EvidenceGraph | None = None

    logger.info(
        "evidence_nexus_initialized",
        storage_path=str(self.storage_path),
        persistence=enable_persistence,
    )

def process(
    self,
    method_outputs: dict[str, Any],
    question_context: dict[str, Any],
    contract: dict[str, Any],
    signal_pack: Any | None = None,
) -> dict[str, Any]:
    """
    Process method outputs into complete answer.

    This is the main entry point that replaces:
    - EvidenceAssembler.assemble()
    - EvidenceValidator.validate()
    - EvidenceRegistry.record_evidence()

    Args:
        method_outputs: Raw outputs from executor methods
        question_context: Question context with expected_elements
        contract: Full v3 contract
        signal_pack: Optional signal pack for provenance

    Returns:
        Complete result dict with:
        - evidence:  Assembled evidence (legacy compatible)
```

```
            - validation: Validation results (legacy compatible)
            - trace: Execution trace (legacy compatible)
            - synthesized_answer: New narrative answer
            - human_readable_output: Formatted answer string
            - graph_statistics: Graph metrics
        """
        start_time = time.time()

        # 1. Build evidence graph from method outputs
        graph = self._build_graph_from_outputs(
            method_outputs, question_context, contract, signal_pack
        )
        self._graph = graph

        # 2. Infer relationships between evidence nodes
        self._infer_relationships(graph, contract)

        # 3. Run belief propagation
        beliefs = graph.compute_belief_propagation()

        # 4. Validate graph
        validation_report = self.validation_engine.validate(graph, contract)

        # 5. Synthesize narrative answer
        synthesized = self.narrative_synthesizer.synthesize(
            graph, question_context, validation_report, contract
        )

        # 6. Persist if enabled
        if self.enable_persistence:
            self._persist_graph(graph)

        # 7. Build legacy-compatible evidence dict
        legacy_evidence = self._build_legacy_evidence(graph, beliefs)
        legacy_validation = self._build_legacy_validation(validation_report)
        legacy_trace = self._build_legacy_trace(graph, signal_pack)

        # ------------------------------------------------------------------
        # SISAS utility / consumption proof (if injected by executor)
        # ------------------------------------------------------------------
        tracker = question_context.get("consumption_tracker")
        if tracker is not None:
            try:
                # ConsumptionTracker provides a summary and an embedded proof object
                if hasattr(tracker, "get_consumption_summary"):
                                                    legacy_trace["signal_consumption"]  =
tracker.get_consumption_summary()
                if hasattr(tracker, "get_proof"):
                    proof = tracker.get_proof()
                    if hasattr(proof, "get_consumption_proof"):
                                                legacy_trace["signal_consumption_proof"]  =
proof.get_consumption_proof()
            except Exception:
                # Never break processing for telemetry failures
```

```
            pass

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

    logger.info(
        "evidence_nexus_process_complete",
        node_count=graph.node_count,
        edge_count=graph.edge_count,
        is_valid=validation_report.is_valid,
        completeness=synthesized.completeness.value,
        confidence=f"{synthesized.overall_confidence:.2f}",
        processing_time_ms=f"{processing_time_ms:.1f}",
    )

    return {
        # Legacy compatible
        "evidence": legacy_evidence,
        "validation": legacy_validation,
        "trace": legacy_trace,

        # New SOTA outputs
        "synthesized_answer":  synthesized.to_dict(),
        "human_readable_output": synthesized.to_human_readable("markdown"),
        "direct_answer": synthesized.direct_answer,

        # Graph data
        "graph_statistics": graph.get_statistics(),
        "graph_hash": graph.get_graph_hash(),

        # Metrics
        "completeness": synthesized.completeness.value,
        "overall_confidence": synthesized.overall_confidence,
        "calibrated_interval": list(synthesized.calibrated_interval),
        "gaps": synthesized.gaps,
        "contradictions": synthesized.unresolved_contradictions,

        # Processing metadata
        "processing_time_ms": processing_time_ms,
        "nexus_version": "1.0.0",
    }

def _build_graph_from_outputs(
    self,
    method_outputs: dict[str, Any],
    question_context: dict[str, Any],
    contract: dict[str, Any],
    signal_pack: Any | None,
) -> EvidenceGraph:
    """
    Transform method outputs into evidence graph.

    Replaces EvidenceAssembler's merge logic with graph construction.
    """
    graph = EvidenceGraph()
```

```
        # ------------------------------------------------------------------
        # Pattern-derived evidence (contract patterns)
        # ------------------------------------------------------------------
        # v3 executors pass patterns + raw_text into question_context so that
        # patterns add value even when downstream methods don't consume them.
        raw_text = (
            question_context.get("raw_text")
            or question_context.get("text")
            or question_context.get("document_text")
        )
        patterns = (
            question_context.get("patterns")
            or contract.get("question_context", {}).get("patterns", [])
        )
          if isinstance(raw_text, str) and raw_text and isinstance(patterns, list) and
patterns:
            graph.add_nodes(
                self._extract_nodes_from_contract_patterns(
                    raw_text=raw_text,
                    patterns=patterns,
                    question_context=question_context,
                )
            )

        # Get assembly rules from contract
        evidence_assembly = contract.get("evidence_assembly", {})
        assembly_rules = evidence_assembly.get("assembly_rules", [])

        # Process each method output
        for source_key, output in method_outputs.items():
            if source_key.startswith("_"):
                continue  # Skip internal keys like _signal_usage

            nodes = self._extract_nodes_from_output(
                source_key, output, question_context
            )
            graph.add_nodes(nodes)

        # Apply assembly rules to create aggregate nodes
        for rule in assembly_rules:
            target = rule.get("target")
            sources = rule.get("sources", [])
            strategy = rule.get("merge_strategy", "concat")

            aggregate_node = self._create_aggregate_node(
                target, sources, strategy, graph, method_outputs
            )
            if aggregate_node:
                graph.add_node(aggregate_node)

        # Add signal provenance if available
        if signal_pack is not None:
            provenance_node = self._create_provenance_node(signal_pack)
```

```
            graph.add_node(provenance_node)

        return graph

    def _extract_nodes_from_contract_patterns(
        self,
        *,
        raw_text: str,
        patterns: list[Any],
        question_context: dict[str, Any],
        max_matches_per_pattern: int = 5,
    ) -> list[EvidenceNode]:
        """Create evidence nodes from v3 contract patterns.

        Goal: patterns contribute to evidence/scoring even if methods ignore them.

        This is intentionally conservative:
        - Only regex/literal matching (NER_OR_REGEX treated as regex fallback)
        - Caps matches per pattern for determinism and bounded output
        """
        nodes: list[EvidenceNode] = []
        qid = str(question_context.get("question_id") or "")
        document_context = question_context.get("document_context")
        if not isinstance(document_context, dict):
            document_context = {}

        # Optional SISAS consumption tracker (utility measurement + proof chain)
        tracker = question_context.get("consumption_tracker")

        # Optional context-aware filtering from SISAS
        filtered_patterns = patterns
        context_filter_stats: dict[str, int] | None = None
        if document_context:
            try:
                                                                                from
cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_context_scoper  import
(
                    filter_patterns_by_context,
                )

                # filter_patterns_by_context expects list[dict], ignore non-dicts
                dict_patterns = [p for p in patterns if isinstance(p, dict)]
                filtered_patterns, context_filter_stats = filter_patterns_by_context(
                    dict_patterns, document_context
                )
            except Exception:
                filtered_patterns = patterns
                context_filter_stats = None

        def _to_flags(flags_value: Any) -> int:
            if not flags_value:
                return re.IGNORECASE | re.MULTILINE
            if isinstance(flags_value, int):
                return flags_value
```

```
            if isinstance(flags_value, str):
                f = 0
                if "i" in flags_value:
                    f |= re.IGNORECASE
                if "m" in flags_value:
                    f |= re.MULTILINE
                if "s" in flags_value:
                    f |= re.DOTALL
                return f or (re.IGNORECASE | re.MULTILINE)
            return re.IGNORECASE | re.MULTILINE

        def _map_category_to_evidence_type(category: str) -> EvidenceType:
            cat = (category or "").upper()
            if cat == "INDICADOR" or cat == "UNIDAD_MEDIDA":
                return EvidenceType.INDICATOR_NUMERIC
            if cat == "TEMPORAL":
                return EvidenceType.TEMPORAL_SERIES
            if cat == "FUENTE_OFICIAL":
                return EvidenceType.OFFICIAL_SOURCE
            if cat == "TERRITORIAL":
                return EvidenceType.TERRITORIAL_COVERAGE
            if cat.startswith("CAUSAL"):
                return EvidenceType.CAUSAL_LINK
            # Default: treat as method-output-like evidence with tags
            return EvidenceType.METHOD_OUTPUT

        for pat in filtered_patterns:
            if not isinstance(pat, dict):
                continue

                        pattern_id = str(pat.get("id") or pat.get("pattern_id") or
pat.get("pattern_ref") or "")
            category = str(pat.get("category") or "GENERAL")
            match_type = str(pat.get("match_type") or "REGEX").upper()
            pattern_str = pat.get("pattern")

            # If pattern_ref exists but pattern is missing, we still emit a trace node
            # so the deficiency is visible in downstream telemetry.
            if not isinstance(pattern_str, str) or not pattern_str.strip():
                node = EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
                    content={
                        "pattern_id": pattern_id,
                        "pattern_ref": pat.get("pattern_ref"),
                        "category": category,
                        "match_type": match_type,
                        "matches": [],
                        "note": "pattern_missing_in_contract",
                        "question_id": qid,
                    },
                    confidence=0.1,
                    source_method="contract.patterns",
                    tags=["contract_pattern", "pattern_missing"],
                )
```

```
                nodes.append(node)
                continue

            flags_int = _to_flags(pat.get("flags"))
            matches: list[str] = []
            try:
                if match_type == "LITERAL":
                            # Simple containment; return matched literal as-is once per
occurrence (bounded).
                    literal = pattern_str
                    start = 0
                    while len(matches) < max_matches_per_pattern:
                        idx = raw_text.lower().find(literal.lower(), start)
                        if idx < 0:
                            break
                        matches.append(raw_text[idx : idx + len(literal)])
                        start = idx + len(literal)
                else:
                    compiled = re.compile(pattern_str, flags_int)
                    for m in compiled.finditer(raw_text):
                        matches.append(m.group(0))
                        if len(matches) >= max_matches_per_pattern:
                            break
            except Exception:
                    # Invalid regex patterns should not crash pipeline; emit diagnostic
node.
                node = EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
                    content={
                        "pattern_id": pattern_id,
                        "pattern_ref": pat.get("pattern_ref"),
                        "category": category,
                        "match_type": match_type,
                        "pattern": pattern_str,
                        "matches": [],
                        "note": "pattern_compile_or_match_failed",
                        "question_id": qid,
                    },
                    confidence=0.1,
                    source_method="contract.patterns",
                    tags=["contract_pattern", "pattern_error"],
                )
                nodes.append(node)
                continue

            # Record consumption proof (if tracker injected)
            if tracker is not None and matches:
                try:
                       # Track up to the same cap; produced_evidence=True because we are
emitting nodes
                    for mtxt in matches:
                                # ConsumptionTracker API: record_pattern_match(pattern,
text_segment, produced_evidence)
                        if hasattr(tracker, "record_pattern_match"):
```

```
                        tracker.record_pattern_match(
                            pattern={"id": pattern_id, "pattern": pattern_str},
                            text_segment=str(mtxt),
                            produced_evidence=True,
                        )
                except Exception:
                    # Never break execution for tracking failures
                    pass

            confidence_weight = pat.get("confidence_weight")
            try:
                confidence = float(confidence_weight) if confidence_weight is not None
else 0.6
            except Exception:
                confidence = 0.6
            if not matches:
                confidence = min(confidence, 0.35)

            node = EvidenceNode.create(
                evidence_type=_map_category_to_evidence_type(category),
                content={
                    "pattern_id": pattern_id,
                    "pattern_ref": pat.get("pattern_ref"),
                    "category": category,
                    "match_type": match_type,
                    "pattern": pattern_str,
                    "matches": matches,
                    "match_count": len(matches),
                    "question_id": qid,
                },
                confidence=max(0.0, min(1.0, confidence)),
                source_method="contract.patterns",
                tags=["contract_pattern", category.lower()],
            )
            nodes.append(node)

        # Emit a lightweight stats node for context filtering and utility accounting
        try:
            total_injected = len([p for p in patterns if isinstance(p, dict)])
            total_considered = len([p for p in filtered_patterns if isinstance(p,
dict)])
            matched_patterns = 0
            for n in nodes:
                if isinstance(n.content, dict) and int(n.content.get("match_count", 0)
or 0) > 0:
                    matched_patterns += 1
            waste_ratio = (
                1.0 - (matched_patterns / total_considered)
                if total_considered > 0
                else 1.0
            )
            nodes.append(
                EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
```

```
                content={
                    "provenance_type": "contract_pattern_utility",
                    "question_id": qid,
                    "patterns_injected": total_injected,
                    "patterns_considered": total_considered,
                    "patterns_matched": matched_patterns,
                    "waste_ratio": round(float(waste_ratio), 4),
                    "context_filter_stats": context_filter_stats,
                },
                confidence=1.0,
                source_method="contract.patterns.utility",
                tags=["contract_pattern", "utility"],
            )
        )
    except Exception:
        pass

    return nodes

def _extract_nodes_from_output(
    self,
    source_key: str,
    output: Any,
    question_context: dict[str, Any],
) -> list[EvidenceNode]:
    """Extract evidence nodes from a single method output."""
    nodes = []

    if output is None:
        return nodes

    # Handle list outputs (multiple evidence items)
    if isinstance(output, list):
        for idx, item in enumerate(output):
            node = self._item_to_node(
                item,
                source_method=source_key,
                index=idx,
            )
            if node:
                nodes.append(node)

    # Handle dict outputs (structured evidence)
    elif isinstance(output, dict):
        # Check if it's a single evidence item or container
        if "elements" in output:
            # Container with elements list
            for idx, item in enumerate(output.get("elements", [])):
                node = self._item_to_node(
                    item,
                    source_method=source_key,
                    index=idx,
                )
                if node:
```

```
                    nodes.append(node)
            else:
                # Single evidence item
                node = self._item_to_node(
                    output,
                    source_method=source_key,
                )
                if node:
                    nodes.append(node)

        # Handle scalar outputs
        else:
            node = EvidenceNode.create(
                evidence_type=EvidenceType.METHOD_OUTPUT,
                content={"value": output, "source":  source_key},
                confidence=0.7,  # Default for raw method output
                source_method=source_key,
            )
            nodes.append(node)

        return nodes

    def _item_to_node(
        self,
        item: Any,
        source_method: str,
        index: int | None = None,
    ) -> EvidenceNode | None:
        """Convert a single item to evidence node."""
        if item is None:
            return None

        if isinstance(item, dict):
            # Extract type
            item_type = item.get("type", item.get("evidence_type", ""))
            try:
                ev_type = EvidenceType(item_type)
            except ValueError:
                ev_type = EvidenceType.METHOD_OUTPUT

            # Extract confidence
            confidence = item.get("confidence", item.get("score", 0.7))
            if isinstance(confidence, str):
                try:
                    confidence = float(confidence.strip("%")) / 100
                except (ValueError, TypeError):
                    confidence = 0.7

            # Extract document location
            doc_loc = item.get("page", item.get("location", item.get("section")))
            if doc_loc is not None:
                doc_loc = str(doc_loc)

            return EvidenceNode.create(
```

```
                evidence_type=ev_type,
                content=item,
                confidence=float(confidence),
                source_method=source_method,
                document_location=doc_loc,
                tags=frozenset(item.get("tags", [])),
            )

        # Non-dict item
        return EvidenceNode.create(
            evidence_type=EvidenceType.METHOD_OUTPUT,
            content={"value": item, "index": index},
            confidence=0.6,
            source_method=source_method,
        )

    def _create_aggregate_node(
        self,
        target: str,
        sources:  list[str],
        strategy:  str,
        graph: EvidenceGraph,
        method_outputs: dict[str, Any],
    ) -> EvidenceNode | None:
        """Create aggregate node from assembly rule."""
        # Collect source values
        values = []
        parent_ids = []

        for source in sources:
            # Resolve dotted path
            value = self._resolve_path(source, method_outputs)
            if value is not None:
                values.append(value)
                # Find corresponding nodes
                for node in graph._nodes.values():
                       # Source keys in method_outputs are dotted paths (no space after
'.')
                    if node.source_method == source.split(".")[0]:
                        parent_ids.append(node.node_id)

        if not values:
            return None

        # Apply merge strategy
        merged_value = self._apply_merge_strategy(values, strategy)

        return EvidenceNode.create(
            evidence_type=EvidenceType.AGGREGATED,
            content={
                "target": target,
                "strategy": strategy,
                "sources":  sources,
                "value": merged_value,
```

```
        },
        confidence=0.8,  # Aggregated confidence
        source_method=f"aggregate:{target}",
        parent_ids=parent_ids[: 10],  # Limit parents
    )

def _resolve_path(self, path: str, data: dict[str, Any]) -> Any:
    """Resolve dotted path in data structure."""
    parts = path.split(".")
    current = data

    for part in parts:
        if isinstance(current, dict) and part in current:
            current = current[part]
        else:
            return None

    return current

def _apply_merge_strategy(
    self,
    values: list[Any],
    strategy: str,
) -> Any:
    """Apply merge strategy to values."""
    if not values:
        return None

    if strategy == "first":
        return values[0]
    elif strategy == "last":
        return values[-1]
    elif strategy == "concat":
        result = []
        for v in values:
            if isinstance(v, list):
                result.extend(v)
            else:
                result.append(v)
        return result
    elif strategy == "mean":
        numeric = [float(v) for v in values if self._is_numeric(v)]
        return statistics.mean(numeric) if numeric else None
    elif strategy == "max":
        numeric = [float(v) for v in values if self._is_numeric(v)]
        return max(numeric) if numeric else None
    elif strategy == "min":
        numeric = [float(v) for v in values if self._is_numeric(v)]
        return min(numeric) if numeric else None
    elif strategy == "weighted_mean":
        numeric = [float(v) for v in values if self._is_numeric(v)]
        return statistics.mean(numeric) if numeric else None
    elif strategy == "majority":
        from collections import Counter
```

```
            counts = Counter(str(v) for v in values)
            return counts.most_common(1)[0][0] if counts else None
        else:
            return values[0]  # Default to first

    @staticmethod
    def _is_numeric(value: Any) -> bool:
        """Check if value is numeric."""
        if isinstance(value, bool):
            return False
        try:
            float(value)
            return True
        except (TypeError, ValueError):
            return False

    def _create_provenance_node(self, signal_pack: Any) -> EvidenceNode:
        """Create provenance node from signal pack."""
          pack_id = getattr(signal_pack, "id", None) or getattr(signal_pack, "pack_id",
"unknown")
        policy_area = getattr(signal_pack, "policy_area", None)
        version = getattr(signal_pack, "version", "unknown")

        return EvidenceNode.create(
            evidence_type=EvidenceType.METHOD_OUTPUT,
            content={
                "provenance_type": "signal_pack",
                "pack_id": pack_id,
                "policy_area": str(policy_area) if policy_area else None,
                "version": version,
            },
            confidence=1.0,  # Provenance is certain
            source_method="signal_registry",
            tags=frozenset(["provenance", "signal_pack"]),
        )

    def _infer_relationships(
        self,
        graph: EvidenceGraph,
        contract:  dict[str, Any],
    ) -> None:
        """
        Infer relationships between evidence nodes.

        Uses:
        - Type compatibility for SUPPORTS edges
        - Temporal ordering for PRECEDES edges
        - Content similarity for CORRELATES edges
        - Contradiction detection for CONTRADICTS edges
        """
        nodes = list(graph._nodes.values())

        # Infer DERIVES_FROM from parent_ids
        for node in nodes:
```

```
            for parent_id in node.parent_ids:
                if parent_id in graph._nodes:
                    edge = EvidenceEdge.create(
                        source_id=parent_id,
                        target_id=node.node_id,
                        relation_type=RelationType.DERIVES_FROM,
                        weight=0.9,
                        confidence=0.95,
                    )
                    try:
                        graph.add_edge(edge)
                    except ValueError:
                        pass  # Skip if would create cycle

        # Infer SUPPORTS between related types
        support_pairs = [
            (EvidenceType.OFFICIAL_SOURCE, EvidenceType.INDICATOR_NUMERIC),
            (EvidenceType.INDICATOR_NUMERIC, EvidenceType.GOAL_TARGET),
            (EvidenceType.BUDGET_AMOUNT, EvidenceType.POLICY_INSTRUMENT),
        ]

        for source_type, target_type in support_pairs:
            source_nodes = graph.get_nodes_by_type(source_type)
            target_nodes = graph.get_nodes_by_type(target_type)

            for sn in source_nodes[: 5]:  # Limit to prevent explosion
                for tn in target_nodes[:5]:
                    if sn.node_id != tn.node_id:
                        edge = EvidenceEdge.create(
                            source_id=sn.node_id,
                            target_id=tn.node_id,
                            relation_type=RelationType.SUPPORTS,
                            weight=0.6,
                            confidence=0.7,
                        )
                        try:
                            graph.add_edge(edge)
                        except ValueError:
                              pass  # Skip if adding SUPPORTS edge would create cycle or
is invalid

    def _persist_graph(self, graph: EvidenceGraph) -> None:
        """Persist graph to storage."""
        if not self.enable_persistence:
            return

        try:
            self.storage_path.parent.mkdir(parents=True, exist_ok=True)

            with open(self.storage_path, "a", encoding="utf-8") as f:
                # Write summary record
                record = {
                    "timestamp":  time.time(),
                    "graph_hash": graph.get_graph_hash(),
```

```
                "node_count": graph.node_count,
                "edge_count": graph.edge_count,
                "statistics": graph.get_statistics(),
            }
            f.write(json.dumps(record, separators=(",", ":")) + "\n")

        logger.debug("graph_persisted", path=str(self.storage_path))

    except Exception as e:
        logger.error("graph_persistence_failed", error=str(e))

def _build_legacy_evidence(
    self,
    graph: EvidenceGraph,
    beliefs: dict[EvidenceID, float],
) -> dict[str, Any]:
    """Build legacy-compatible evidence dict."""
    elements = []
    by_type:  dict[str, list[dict]] = defaultdict(list)
    confidences = []

    for node in graph._nodes.values():
        elem = {
            "element_id": node.node_id[: 12],
            "type": node.evidence_type.value,
            "value": self._extract_value(node.content),
            "confidence": node.confidence,
            "belief":  beliefs.get(node.node_id, node.confidence),
            "source_method": node.source_method,
        }
        elements.append(elem)
        by_type[node.evidence_type.value].append(elem)
        confidences.append(node.confidence)

    return {
        "elements": elements,
        "elements_found_count": len(elements),
        "by_type": {k: len(v) for k, v in by_type.items()},
        "confidence_scores": {
            "mean": statistics.mean(confidences) if confidences else 0.0,
            "min": min(confidences) if confidences else 0.0,
            "max": max(confidences) if confidences else 0.0,
        },
        "graph_hash": graph.get_graph_hash()[: 16],
    }

def _extract_value(self, content: dict[str, Any]) -> Any:
    """Extract primary value from content dict."""
    for key in ["value", "text", "description", "name", "indicator"]:
        if key in content:
            return content[key]
    return content

def _build_legacy_validation(
```

```
        self,
        report: ValidationReport,
    ) -> dict[str, Any]:
        """Build legacy-compatible validation dict."""
        return {
            "valid":  report.is_valid,
            "passed":  report.is_valid,
                   "errors":  [f.to_dict()  for  f  in  report.findings  if  f.severity  in
(ValidationSeverity.CRITICAL, ValidationSeverity.ERROR)],
                  "warnings":  [f.to_dict()  for  f  in  report.findings  if  f.severity  ==
ValidationSeverity.WARNING],
            "critical_count": report.critical_count,
            "error_count": report.error_count,
            "warning_count": report.warning_count,
            "consistency_score": report.consistency_score,
            "graph_integrity": report.graph_integrity,
        }

    def _build_legacy_trace(
        self,
        graph: EvidenceGraph,
        signal_pack: Any | None,
    ) -> dict[str, Any]:
        """Build legacy-compatible trace dict."""
        trace = {
            "graph_statistics": graph.get_statistics(),
            "hash_chain_length": len(graph._hash_chain),
            "processing_timestamp": time.time(),
        }

        # Contract pattern utility summary (always available if pattern extraction ran)
        try:
            utility_nodes = graph.get_nodes_by_source("contract.patterns.utility")
            if utility_nodes:
                # Keep last node (single) as authoritative
                node = utility_nodes[-1]
                if isinstance(node.content, dict):
                    trace["pattern_utility"] = {
                        "patterns_injected": node.content.get("patterns_injected"),
                        "patterns_considered": node.content.get("patterns_considered"),
                        "patterns_matched": node.content.get("patterns_matched"),
                        "waste_ratio": node.content.get("waste_ratio"),
                                                              "context_filter_stats":
node.content.get("context_filter_stats"),
                    }
        except Exception:
            pass

        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":  str(getattr(signal_pack, "policy_area", None)),
                "version": getattr(signal_pack, "version", "unknown"),
```

```
            }

        return trace

    # ------------------------------------------------------------------------
    # Public Query Interface
    # ------------------------------------------------------------------------

    def get_current_graph(self) -> EvidenceGraph | None:
        """Get current session graph."""
        return self._graph

    def query_by_type(self, evidence_type: EvidenceType) -> list[EvidenceNode]:
        """Query nodes by type from current graph."""
        if self._graph is None:
            return []
        return self._graph.get_nodes_by_type(evidence_type)

    def query_by_source(self, source_method: str) -> list[EvidenceNode]:
        """Query nodes by source method from current graph."""
        if self._graph is None:
            return []
        return self._graph.get_nodes_by_source(source_method)

    def get_statistics(self) -> dict[str, Any]:
        """Get current graph statistics."""
        if self._graph is None:
            return {"error": "No graph in current session"}
        return self._graph.get_statistics()


# ============================================================================
# FACTORY AND CONVENIENCE FUNCTIONS
# ============================================================================

# Global instance (singleton pattern for registry compatibility)
_global_nexus: EvidenceNexus | None = None


def get_global_nexus() -> EvidenceNexus:
    """Get or create global EvidenceNexus instance."""
    global _global_nexus
    if _global_nexus is None:
        _global_nexus = EvidenceNexus()
    return _global_nexus


def process_evidence(
    method_outputs: dict[str, Any],
    question_context: dict[str, Any],
    contract: dict[str, Any],
    signal_pack: Any | None = None,
) -> dict[str, Any]:
    """
```

```
    Convenience function for one-shot evidence processing.

    This replaces the typical pattern of:
        assembled = EvidenceAssembler.assemble(...)
        validation = EvidenceValidator.validate(...)
        registry.record_evidence(...)

    With:
        result = process_evidence(...)
    """
    nexus = get_global_nexus()
    return nexus.process(
        method_outputs=method_outputs,
        question_context=question_context,
        contract=contract,
        signal_pack=signal_pack,
    )


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

__all__ = [
    # Core types
    "EvidenceType",
    "RelationType",
    "ValidationSeverity",
    "AnswerCompleteness",
    "NarrativeSection",

    # Data structures
    "EvidenceNode",
    "EvidenceEdge",
    "ValidationFinding",
    "ValidationReport",
    "Citation",
    "NarrativeBlock",
    "SynthesizedAnswer",

    # Graph
    "EvidenceGraph",

    # Engines
    "ValidationEngine",
    "NarrativeSynthesizer",

    # Main class
    "EvidenceNexus",

    # Factory functions
    "get_global_nexus",
    "process_evidence",
]
```

```
src/farfan_pipeline/phases/Phase_two/executor_config.py

"""
ExecutorConfig: Runtime parametrization for executors (HOW we execute).

CRITICAL SEPARATION:
- This file contains ONLY runtime parameters (timeout, retry, etc.)
- NO calibration values (quality scores, fusion weights) are stored here
- Calibration data (WHAT quality) is loaded from:
                                                                                *
src/cross_cutting_infrastructure/capaz_calibration_parmetrization/calibration/COHORT_202
4_intrinsic_calibration.json
  * canonic_questionnaire_central/questionnaire_monolith.json

Loading hierarchy (highest to lowest priority):
1. CLI arguments (--timeout-s=120)
2. Environment variables (FARFAN_TIMEOUT_S=120)
3. Environment file (system/config/environments/{env}.json)
4. Executor config file (executor_configs/{executor_id}.json)
5. Conservative defaults

See CALIBRATION_VS_PARAMETRIZATION.md for complete specification.
"""

from __future__ import annotations

import json
import os
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Dict, Optional


@dataclass
class ExecutorConfig:
    """
    Runtime configuration for executor execution (HOW parameters only).

    This dataclass contains ONLY execution parameters that control HOW
    executors run, NOT calibration values that define WHAT quality we measure.

    Loading Hierarchy:
        CLI args > ENV vars > environment file > executor config file > defaults

    Attributes:
        timeout_s: Maximum execution time in seconds
        retry: Number of retry attempts on failure
        temperature: LLM sampling temperature (0.0 = deterministic)
        max_tokens: Maximum LLM output tokens
        memory_limit_mb: Memory limit in megabytes
        enable_profiling: Whether to enable execution profiling
        seed: Random seed for reproducibility
        extra: Additional executor-specific parameters
    """
```

```
    timeout_s: float | None = None
    retry: int | None = None
    temperature: float | None = None
    max_tokens: int | None = None
    memory_limit_mb: int | None = None
    enable_profiling: bool = True
    seed: int | None = None
    extra: dict[str, Any] | None = None

    def __post_init__(self) -> None:
        if self.timeout_s is not None and self.timeout_s <= 0:
            raise ValueError("timeout_s must be positive when provided")
        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")
        if self.temperature is not None and not (0.0 <= self.temperature <= 2.0):
            raise ValueError("temperature must be in range [0.0, 2.0]")
        if self.memory_limit_mb is not None and self.memory_limit_mb <= 0:
            raise ValueError("memory_limit_mb must be positive when provided")

    @classmethod
    def from_dict(cls, config_dict: Dict[str, Any]) -> ExecutorConfig:
        """Create ExecutorConfig from dictionary."""
        valid_fields = {
            "timeout_s", "retry", "temperature", "max_tokens",
            "memory_limit_mb", "enable_profiling", "seed", "extra"
        }
        filtered = {k: v for k, v in config_dict.items() if k in valid_fields}
        return cls(**filtered)

    @classmethod
    def load_from_sources(
        cls,
        executor_id: str,
        environment: str = "production",
        cli_overrides: Optional[Dict[str, Any]] = None
    ) -> ExecutorConfig:
        """
        Load ExecutorConfig from multiple sources with proper hierarchy.

        Loading order (highest to lowest priority):
        1. CLI arguments (passed via cli_overrides)
        2. Environment variables (FARFAN_*)
        3. Environment file (system/config/environments/{env}.json)
        4. Executor config file (executor_configs/{executor_id}.json)
        5. Conservative defaults

        Args:
                                            executor_id:    Executor    identifier    (e.g.,
"D3_Q2_TargetProportionalityAnalyzer")
            environment: Environment name (development, staging, production)
            cli_overrides: CLI argument overrides
```

```
        Returns:
            ExecutorConfig with merged configuration
        """
        config = cls._get_conservative_defaults()

        executor_config = cls._load_executor_config_file(executor_id)
        if executor_config:
            config.update(executor_config)

        env_config = cls._load_environment_file(environment)
        if env_config and "executor" in env_config:
            config.update(env_config["executor"])

        env_vars = cls._load_environment_variables()
        config.update(env_vars)

        if cli_overrides:
            config.update(cli_overrides)

        return cls.from_dict(config)

    @staticmethod
    def _get_conservative_defaults() -> Dict[str, Any]:
        """Get conservative default parameters."""
        return {
            "timeout_s": 300.0,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": 512,
            "enable_profiling": True,
            "seed": 42,
        }

    @staticmethod
    def _load_executor_config_file(executor_id: str) -> Optional[Dict[str, Any]]:
        """Load executor-specific config file."""
        config_file = Path(__file__).parent / "executor_configs" / f"{executor_id}.json"

        if not config_file.exists():
            return None

        try:
            with open(config_file) as f:
                data = json.load(f)
                return data.get("runtime_parameters", {})
        except (json.JSONDecodeError, IOError):
            return None

    @staticmethod
    def _load_environment_file(environment: str) -> Optional[Dict[str, Any]]:
        """Load environment-specific config file."""
          base_path = Path(__file__).parent.parent.parent.parent / "system" / "config" /
```

```
"environments"
        env_file = base_path / f"{environment}.json"

        if not env_file.exists():
            return None

        try:
            with open(env_file) as f:
                return json.load(f)
        except (json.JSONDecodeError, IOError):
            return None

    @staticmethod
    def _load_environment_variables() -> Dict[str, Any]:
        """Load configuration from environment variables."""
        config = {}

        if "FARFAN_TIMEOUT_S" in os.environ:
            config["timeout_s"] = float(os.environ["FARFAN_TIMEOUT_S"])
        if "FARFAN_RETRY" in os.environ:
            config["retry"] = int(os.environ["FARFAN_RETRY"])
        if "FARFAN_TEMPERATURE" in os.environ:
            config["temperature"] = float(os.environ["FARFAN_TEMPERATURE"])
        if "FARFAN_MAX_TOKENS" in os.environ:
            config["max_tokens"] = int(os.environ["FARFAN_MAX_TOKENS"])
        if "FARFAN_MEMORY_LIMIT_MB" in os.environ:
            config["memory_limit_mb"] = int(os.environ["FARFAN_MEMORY_LIMIT_MB"])
        if "FARFAN_SEED" in os.environ:
            config["seed"] = int(os.environ["FARFAN_SEED"])

        return config

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary, excluding None values."""
        return {
            k: v for k, v in {
                "timeout_s": self.timeout_s,
                "retry": self.retry,
                "temperature": self.temperature,
                "max_tokens": self.max_tokens,
                "memory_limit_mb": self.memory_limit_mb,
                "enable_profiling": self.enable_profiling,
                "seed": self.seed,
                "extra": self.extra,
            }.items() if v is not None
        }


__all__ = ["ExecutorConfig"]
```