```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 20
================================================================================
Generated: 2025-12-07T06:17:25.812595
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/processing/aggregation.py
================================================================================

"""
Aggregation Module - Hierarchical Score Aggregation System

This module implements the complete aggregation pipeline for the policy analysis system:
- FASE 4: Dimension aggregation (60 dimensions: 6 Ã\227 10 policy areas)
- FASE 5: Policy area aggregation (10 areas)
- FASE 6: Cluster aggregation (4 MESO questions)
- FASE 7: Macro evaluation (1 holistic question)

Requirements:
- Validation of weights, thresholds, and hermeticity
- Comprehensive logging and abortability at each level
- No strategic simplification
- Full alignment with monolith specifications
- Uses canonical notation for dimension and policy area validation

Architecture:
- DimensionAggregator: Aggregates 5 micro questions â\206\222 1 dimension score
- AreaPolicyAggregator: Aggregates 6 dimension scores â\206\222 1 area score
- ClusterAggregator: Aggregates multiple area scores â\206\222 1 cluster score
- MacroAggregator: Aggregates all cluster scores â\206\222 1 holistic evaluation
"""

from __future__ import annotations

import logging
from collections import defaultdict
from dataclasses import dataclass, field
from typing import TYPE_CHECKING, Any, TypeVar
from farfan_pipeline.core.calibration.parameter_loader import get_parameter_loader
from farfan_pipeline.core.calibration.decorators import calibrated_method
from farfan_pipeline.core.parameters import ParameterLoaderV2

# SOTA imports
from farfan_pipeline.processing.aggregation_provenance import (
    AggregationDAG,
    ProvenanceNode,
)
from farfan_pipeline.processing.uncertainty_quantification import (
    BootstrapAggregator,
    UncertaintyMetrics,
    aggregate_with_uncertainty,
)
from farfan_pipeline.processing.choquet_adapter import (
```

```
    ChoquetProcessingAdapter,
    create_default_choquet_adapter,
)

if TYPE_CHECKING:
    from collections.abc import Callable, Iterable

T = TypeVar('T')


@dataclass(frozen=True)
class AggregationSettings:
    """Resolved aggregation settings derived from the questionnaire monolith."""

    dimension_group_by_keys: list[str]
    area_group_by_keys: list[str]
    cluster_group_by_keys: list[str]
    dimension_question_weights: dict[str, dict[str, float]]
    policy_area_dimension_weights: dict[str, dict[str, float]]
    cluster_policy_area_weights: dict[str, dict[str, float]]
    macro_cluster_weights: dict[str, float]
    dimension_expected_counts: dict[tuple[str, str], int]
    area_expected_dimension_counts: dict[str, int]

    @classmethod
    def from_monolith(cls, monolith: dict[str, Any] | None) -> AggregationSettings:
        """Build aggregation settings from canonical questionnaire data."""
        if not monolith:
            return cls(
                dimension_group_by_keys=["policy_area", "dimension"],
                area_group_by_keys=["area_id"],
                cluster_group_by_keys=["cluster_id"],
                dimension_question_weights={},
                policy_area_dimension_weights={},
                cluster_policy_area_weights={},
                macro_cluster_weights={},
                dimension_expected_counts={},
                area_expected_dimension_counts={},
            )

        blocks = monolith.get("blocks", {})
        niveles = blocks.get("niveles_abstraccion", {})
        policy_areas = niveles.get("policy_areas", [])
        clusters = niveles.get("clusters", [])
        micro_questions = blocks.get("micro_questions", [])

        aggregation_block = (
            monolith.get("aggregation")
            or blocks.get("aggregation")
            or monolith.get("rubric", {}).get("aggregation")
            or {}
        )

        # Map question_id â\206\222 base_slot for later normalization
        question_slot_lookup: dict[str, str] = {}
        dimension_slot_map: dict[str, set[str]] = defaultdict(set)
```

```
        dimension_expected_counts: dict[tuple[str, str], int] = defaultdict(int)

        for question in micro_questions:
            qid = question.get("question_id")
            dim_id = question.get("dimension_id") or question.get("dimension")
            area_id = question.get("policy_area_id") or question.get("policy_area")
            base_slot = question.get("base_slot")

            if dim_id and qid and not base_slot:
                base_slot = f"{dim_id}-{qid}"

            if qid and base_slot:
                question_slot_lookup[qid] = base_slot
                dimension_slot_map[dim_id].add(base_slot)

            if area_id and dim_id:
                dimension_expected_counts[(area_id, dim_id)] += 1

        area_expected_dimension_counts: dict[str, int] = {}
        for area in policy_areas:
            area_id = area.get("policy_area_id") or area.get("id")
            if not area_id:
                continue
            dims = area.get("dimension_ids") or []
            area_expected_dimension_counts[area_id] = len(dims)

        group_by_block = aggregation_block.get("group_by_keys") or {}
        dimension_group_by_keys = cls._coerce_str_list(
            group_by_block.get("dimension"),
            fallback=["policy_area", "dimension"],
        )
        area_group_by_keys = cls._coerce_str_list(
            group_by_block.get("area"),
            fallback=["area_id"],
        )
        cluster_group_by_keys = cls._coerce_str_list(
            group_by_block.get("cluster"),
            fallback=["cluster_id"],
        )

        dimension_question_weights = cls._build_dimension_weights(
            aggregation_block.get("dimension_question_weights") or {},
            question_slot_lookup,
            dimension_slot_map,
        )
        policy_area_dimension_weights = cls._build_area_dimension_weights(
            aggregation_block.get("policy_area_dimension_weights") or {},
            policy_areas,
        )
        cluster_policy_area_weights = cls._build_cluster_weights(
            aggregation_block.get("cluster_policy_area_weights") or {},
            clusters,
        )
        macro_cluster_weights = cls._build_macro_weights(
            aggregation_block.get("macro_cluster_weights") or {},
            clusters,
```

```
        )

        return cls(
            dimension_group_by_keys=dimension_group_by_keys,
            area_group_by_keys=area_group_by_keys,
            cluster_group_by_keys=cluster_group_by_keys,
            dimension_question_weights=dimension_question_weights,
            policy_area_dimension_weights=policy_area_dimension_weights,
            cluster_policy_area_weights=cluster_policy_area_weights,
            macro_cluster_weights=macro_cluster_weights,
            dimension_expected_counts=dict(dimension_expected_counts),
            area_expected_dimension_counts=area_expected_dimension_counts,
        )

    @staticmethod
    def _coerce_str_list(value: Any, *, fallback: list[str]) -> list[str]:
        if isinstance(value, list) and all(isinstance(item, str) for item in value):
            return value or fallback
        return fallback

    @staticmethod
    def _normalize_weights(weight_map: dict[str, float]) -> dict[str, float]:
        if not weight_map:
            return {}
        # Discard negative weights and normalize remaining ones
        positive_map = {k: float(v) for k, v in weight_map.items() if isinstance(v, (float, int)) and float(v) >= 0.0}
        if not positive_map:
            equal = 1.0 / len(weight_map)
            return {k: equal for k in weight_map}
        total = sum(positive_map.values())
        if total <= 0:
            equal = 1.0 / len(positive_map)
            return {k: equal for k in positive_map}
        return {k: value / total for k, value in positive_map.items()}

    @classmethod
    def _build_dimension_weights(
        cls,
        raw_weights: dict[str, dict[str, Any]],
        question_slot_lookup: dict[str, str],
        dimension_slot_map: dict[str, set[str]],
    ) -> dict[str, dict[str, float]]:
        dimension_weights: dict[str, dict[str, float]] = {}
        if raw_weights:
            for dim_id, weights in raw_weights.items():
                resolved: dict[str, float] = {}
                for qid, weight in weights.items():
                    slot = question_slot_lookup.get(qid, qid)
                    try:
                        resolved[slot] = float(weight)
                    except (TypeError, ValueError):
                        continue
                if resolved:
                    dimension_weights[dim_id] = cls._normalize_weights(resolved)

        if not dimension_weights:
```

```
            for dim_id, slots in dimension_slot_map.items():
                if not slots:
                    continue
                equal = 1.0 / len(slots)
                dimension_weights[dim_id] = {slot: equal for slot in slots}

        return dimension_weights

    @classmethod
    def _build_area_dimension_weights(
        cls,
        raw_weights: dict[str, dict[str, Any]],
        policy_areas: list[dict[str, Any]],
    ) -> dict[str, dict[str, float]]:
        area_weights: dict[str, dict[str, float]] = {}
        if raw_weights:
            for area_id, weights in raw_weights.items():
                resolved: dict[str, float] = {}
                for dim_id, value in weights.items():
                    try:
                        resolved[dim_id] = float(value)
                    except (TypeError, ValueError):
                        continue
                if resolved:
                    area_weights[area_id] = cls._normalize_weights(resolved)

        if not area_weights:
            for area in policy_areas:
                area_id = area.get("policy_area_id") or area.get("id")
                dims = area.get("dimension_ids") or []
                if not area_id or not dims:
                    continue
                equal = 1.0 / len(dims)
                area_weights[area_id] = {dim: equal for dim in dims}

        return area_weights

    @classmethod
    def _build_cluster_weights(
        cls,
        raw_weights: dict[str, dict[str, Any]],
        clusters: list[dict[str, Any]],
    ) -> dict[str, dict[str, float]]:
        cluster_weights: dict[str, dict[str, float]] = {}
        if raw_weights:
            for cluster_id, weights in raw_weights.items():
                resolved: dict[str, float] = {}
                for area_id, value in weights.items():
                    try:
                        resolved[area_id] = float(value)
                    except (TypeError, ValueError):
                        continue
                if resolved:
                    cluster_weights[cluster_id] = cls._normalize_weights(resolved)

        if not cluster_weights:
```

```
            for cluster in clusters:
                cluster_id = cluster.get("cluster_id")
                area_ids = cluster.get("policy_area_ids") or []
                if not cluster_id or not area_ids:
                    continue
                equal = 1.0 / len(area_ids)
                cluster_weights[cluster_id] = {area_id: equal for area_id in area_ids}

        return cluster_weights

    @classmethod
    def _build_macro_weights(
        cls,
        raw_weights: dict[str, Any],
        clusters: list[dict[str, Any]],
    ) -> dict[str, float]:
        if raw_weights:
            resolved = {}
            for cluster_id, weight in raw_weights.items():
                try:
                    resolved[cluster_id] = float(weight)
                except (TypeError, ValueError):
                    continue
            normalized = cls._normalize_weights(resolved)
            if normalized:
                return normalized

        cluster_ids = [cluster.get("cluster_id") for cluster in clusters if cluster.get("cluster_id")]
        if not cluster_ids:
            return {}
        equal = 1.0 / len(cluster_ids)
        return {cluster_id: equal for cluster_id in cluster_ids}

def group_by(items: Iterable[T], key_func: Callable[[T], tuple]) -> dict[tuple, list[T]]:
    """
    Groups a sequence of items into a dictionary based on a key function.

    This utility function iterates over a collection, applies a key function to each
    item, and collects items into lists, keyed by the result of the key function.

    The key function must return a tuple. This is because dictionary keys must be
    hashable, and tuples are hashable whereas lists are not. Using a tuple allows
    for grouping by multiple attributes.

    If the input iterable `items` is empty, this function will return an empty
    dictionary.

    Example:
        >>> from dataclasses import dataclass
        >>> @dataclass
        ... class Record:
        ...     category: str
        ...     value: int
        ...
        >>> data = [Record("A", 1), Record("B", 2), Record("A", 3)]
        >>> group_by(data, key_func=lambda r: (r.category,))
```

```
        {('A',): [Record(category='A', value=1), Record(category='A', value=3)],
         ('B',): [Record(category='B', value=2)]}

    Args:
        items: An iterable of items to be grouped.
        key_func: A callable that accepts an item and returns a tuple to be
                  used as the grouping key.

    Returns:
        A dictionary where keys are the result of the key function and values are
        lists of items belonging to that group.
    """
    grouped = defaultdict(list)
    for item in items:
        grouped[key_func(item)].append(item)
    return dict(grouped)

def validate_scored_results(results: list[dict[str, Any]]) -> list[ScoredResult]:
    """
    Validates a list of dictionaries and converts them to ScoredResult objects.

    Args:
        results: A list of dictionaries representing scored results.

    Returns:
        A list of ScoredResult objects.

    Raises:
        ValidationError: If any of the dictionaries are invalid.
    """
    validated_results = []
    required_keys = {
        "question_global": int, "base_slot": str, "policy_area": str, "dimension": str,
        "score": float, "quality_level": str, "evidence": dict, "raw_results": dict
    }
    for i, res_dict in enumerate(results):
        missing_keys = set(required_keys.keys()) - set(res_dict.keys())
        if missing_keys:
            raise ValidationError(
                f"Invalid ScoredResult at index {i}: missing keys {missing_keys}"
            )
        for key, expected_type in required_keys.items():
            if not isinstance(res_dict[key], expected_type):
                raise ValidationError(
                    f"Invalid type for key '{key}' at index {i}. "
                    f"Expected {expected_type}, got {type(res_dict[key])}."
                )
        try:
            validated_results.append(ScoredResult(**res_dict))
        except TypeError as e:
            raise ValidationError(f"Invalid ScoredResult at index {i}: {e}") from e
    return validated_results

# Import canonical notation for validation
try:
    from farfan_pipeline.core.canonical_notation import get_all_dimensions, get_all_policy_areas
```

```
393:     HAS_CANONICAL_NOTATION = True
394: except ImportError:
395:     HAS_CANONICAL_NOTATION = False
396:
397: logger = logging.getLogger(__name__)
398:
399: @dataclass
400: class ScoredResult:
401:     """Represents a single, scored micro-question, forming the input for aggregation."""
402:     question_global: int
403:     base_slot: str
404:     policy_area: str
405:     dimension: str
406:     score: float
407:     quality_level: str
408:     evidence: dict[str, Any]
409:     raw_results: dict[str, Any]
410:
411: @dataclass
412: class DimensionScore:
413:     """
414:     Aggregated score for a single dimension within a policy area.
415:
416:     SOTA Extensions:
417:     - Uncertainty quantification (mean, std, CI)
418:     - Provenance tracking (DAG node ID)
419:     - Aggregation method recording
420:     """
421:     dimension_id: str
422:     area_id: str
423:     score: float
424:     quality_level: str
425:     contributing_questions: list[int]
426:     validation_passed: bool = True
427:     validation_details: dict[str, Any] = field(default_factory=dict)
428:
429:     # SOTA: Uncertainty quantification
430:     score_std: float = 0.0
431:     confidence_interval_95: tuple[float, float] = field(default_factory=lambda: (0.0, 0.0))
432:     epistemic_uncertainty: float = 0.0
433:     aleatoric_uncertainty: float = 0.0
434:
435:     # SOTA: Provenance tracking
436:     provenance_node_id: str = ""
437:     aggregation_method: str = "weighted_average"
438:
439: @dataclass
440: class AreaScore:
441:     """
442:     Aggregated score for a policy area, based on its constituent dimensions.
443:
444:     SOTA Extensions:
445:     - Uncertainty quantification
446:     - Provenance tracking
447:     """
448:     area_id: str
```

```
449:     area_name: str
450:     score: float
451:     quality_level: str
452:     dimension_scores: list[DimensionScore]
453:     validation_passed: bool = True
454:     validation_details: dict[str, Any] = field(default_factory=dict)
455:     cluster_id: str | None = None
456:
457:     # SOTA: Uncertainty quantification
458:     score_std: float = 0.0
459:     confidence_interval_95: tuple[float, float] = field(default_factory=lambda: (0.0, 0.0))
460:
461:     # SOTA: Provenance tracking
462:     provenance_node_id: str = ""
463:     aggregation_method: str = "weighted_average"
464:
465: @dataclass
466: class ClusterScore:
467:     """
468:     Aggregated score for a MESO cluster, based on its policy areas.
469:
470:     SOTA Extensions:
471:     - Uncertainty quantification
472:     - Provenance tracking
473:     """
474:     cluster_id: str
475:     cluster_name: str
476:     areas: list[str]
477:     score: float
478:     coherence: float
479:     variance: float
480:     weakest_area: str | None
481:     area_scores: list[AreaScore]
482:     validation_passed: bool = True
483:     validation_details: dict[str, Any] = field(default_factory=dict)
484:
485:     # SOTA: Uncertainty quantification
486:     score_std: float = 0.0
487:     confidence_interval_95: tuple[float, float] = field(default_factory=lambda: (0.0, 0.0))
488:
489:     # SOTA: Provenance tracking
490:     provenance_node_id: str = ""
491:     aggregation_method: str = "weighted_average"
492:
493: @dataclass
494: class MacroScore:
495:     """Represents the final, holistic macro evaluation score for the entire system."""
496:     score: float
497:     quality_level: str
498:     cross_cutting_coherence: float  # Coherence across all clusters
499:     systemic_gaps: list[str]
500:     strategic_alignment: float
501:     cluster_scores: list[ClusterScore]
502:     validation_passed: bool = True
503:     validation_details: dict[str, Any] = field(default_factory=dict)
504:
```

```
505: class AggregationError(Exception):
506:     """Base exception for aggregation errors."""
507:     pass
508:
509: class ValidationError(AggregationError):
510:     """Raised when validation fails."""
511:     pass
512:
513: class WeightValidationError(ValidationError):
514:     """Raised when weight validation fails."""
515:     pass
516:
517: class ThresholdValidationError(ValidationError):
518:     """Raised when threshold validation fails."""
519:     pass
520:
521: class HermeticityValidationError(ValidationError):
522:     """Raised when hermeticity validation fails."""
523:     pass
524:
525: class CoverageError(AggregationError):
526:     """Raised when coverage requirements are not met."""
527:     pass
528:
529: class DimensionAggregator:
530:     """
531:     Aggregates micro question scores into dimension scores.
532:
533:     Responsibilities:
534:     - Aggregate 5 micro questions (Q1-Q5) per dimension
535:     - Validate weights sum to 1.0
536:     - Apply rubric thresholds
537:     - Ensure coverage (abort if insufficient)
538:     - Provide detailed logging
539:     """
540:
541:     def __init__(
542:         self,
543:         monolith: dict[str, Any] | None = None,
544:         abort_on_insufficient: bool = True,
545:         aggregation_settings: AggregationSettings | None = None,
546:         enable_sota_features: bool = True,
547:     ) -> None:
548:         """
549:         Initialize dimension aggregator.
550:
551:         Args:
552:             monolith: Questionnaire monolith configuration (optional, required for run())
553:             abort_on_insufficient: Whether to abort on insufficient coverage
554:             aggregation_settings: Resolved aggregation settings
555:             enable_sota_features: Enable SOTA features (Choquet, UQ, provenance)
556:
557:         Raises:
558:             ValueError: If monolith is None and required for operations
559:         """
560:         self.monolith = monolith
```

```
        self.abort_on_insufficient = abort_on_insufficient
        self.aggregation_settings = aggregation_settings or AggregationSettings.from_monolith(monolith)
        self.dimension_group_by_keys = (
            self.aggregation_settings.dimension_group_by_keys or ["policy_area", "dimension"]
        )
        self.enable_sota_features = enable_sota_features

        # Extract configuration if monolith provided
        if monolith is not None:
            self.scoring_config = monolith["blocks"]["scoring"]
            self.niveles = monolith["blocks"]["niveles_abstraccion"]
        else:
            self.scoring_config = None
            self.niveles = None

        # SOTA: Initialize provenance DAG
        if self.enable_sota_features:
            self.provenance_dag = AggregationDAG()
            self.bootstrap_aggregator = BootstrapAggregator(n_samples=1000, random_seed=42)
            logger.info("DimensionAggregator initialized with SOTA features enabled")
        else:
            self.provenance_dag = None
            self.bootstrap_aggregator = None
            logger.info("DimensionAggregator initialized (legacy mode)")

        # Validate canonical notation if available
        if HAS_CANONICAL_NOTATION:
            try:
                canonical_dims = get_all_dimensions()
                canonical_areas = get_all_policy_areas()
                logger.info(
                    f"Canonical notation loaded: {len(canonical_dims)} dimensions, "
                    f"{len(canonical_areas)} policy areas"
                )
            except Exception as e:
                logger.warning(f"Could not load canonical notation: {e}")

    @calibrated_method("farfan_core.processing.aggregation.DimensionAggregator.validate_dimension_id")
    def validate_dimension_id(self, dimension_id: str) -> bool:
        """
        Validate dimension ID against canonical notation.

        Args:
            dimension_id: Dimension ID to validate (e.g., "DIM01")

        Returns:
            True if dimension ID is valid

        Raises:
            ValidationError: If dimension ID is invalid and abort_on_insufficient is True
        """
        if not HAS_CANONICAL_NOTATION:
            logger.debug("Canonical notation not available, skipping validation")
            return True

        try:
```

```
            canonical_dims = get_all_dimensions()
            # Check if dimension_id is a valid code
            valid_codes = {info.code for info in canonical_dims.values()}
            if dimension_id in valid_codes:
                return True

            msg = f"Invalid dimension ID: {dimension_id}. Valid codes: {sorted(valid_codes)}"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise ValidationError(msg)
            return False
        except Exception as e:
            logger.warning(f"Could not validate dimension ID: {e}")
            return True  # Don't fail if validation can't be performed

    @calibrated_method("farfan_core.processing.aggregation.DimensionAggregator.validate_policy_area_id")
    def validate_policy_area_id(self, area_id: str) -> bool:
        """
        Validate policy area ID against canonical notation.

        Args:
            area_id: Policy area ID to validate (e.g., "PA01")

        Returns:
            True if policy area ID is valid

        Raises:
            ValidationError: If policy area ID is invalid and abort_on_insufficient is True
        """
        if not HAS_CANONICAL_NOTATION:
            logger.debug("Canonical notation not available, skipping validation")
            return True

        try:
            canonical_areas = get_all_policy_areas()
            if area_id in canonical_areas:
                return True

            msg = f"Invalid policy area ID: {area_id}. Valid codes: {sorted(canonical_areas.keys())}"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise ValidationError(msg)
            return False
        except Exception as e:
            logger.warning(f"Could not validate policy area ID: {e}")
            return True  # Don't fail if validation can't be performed

    @calibrated_method("farfan_core.processing.aggregation.DimensionAggregator.validate_weights")
    def validate_weights(self, weights: list[float]) -> tuple[bool, str]:
        """
        Ensures that a list of weights sums to ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_
L582_47", 1.0) within a small tolerance.

        Args:
            weights: A list of floating-point weights.

```

```
        Returns:
            A tuple containing a boolean indicating validity and a descriptive message.

        Raises:
            WeightValidationError: If `abort_on_insufficient` is True and validation fails.
        """
        if not weights:
            msg = "No weights provided"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise WeightValidationError(msg)
            return False, msg

        weight_sum = sum(weights)
        tolerance = 1e-6

        if abs(weight_sum - ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L603_28", 1.0)) > t
olerance:
            expected_weight = ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L604_81", 1.0)
            msg = f"Weight sum validation failed: sum={weight_sum:.6f}, expected={expected_weight}"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise WeightValidationError(msg)
            return False, msg

        logger.debug(f"Weight validation passed: sum={weight_sum:.6f}")
        return True, "Weights valid"

    def validate_coverage(
        self,
        results: list[ScoredResult],
        expected_count: int = 5
    ) -> tuple[bool, str]:
        """
        Checks if the number of results meets a minimum expectation.

        Args:
            results: A list of ScoredResult objects.
            expected_count: The minimum number of results required.

        Returns:
            A tuple containing a boolean indicating validity and a descriptive message.

        Raises:
            CoverageError: If `abort_on_insufficient` is True and coverage is insufficient.
        """
        actual_count = len(results)

        if actual_count < expected_count:
            msg = (
                f"Coverage validation failed: "
                f"expected {expected_count} questions, got {actual_count}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise CoverageError(msg)
```

```
            return False, msg

        logger.debug(f"Coverage validation passed: {actual_count}/{expected_count} questions")
        return True, "Coverage sufficient"

    def calculate_weighted_average(
        self,
        scores: list[float],
        weights: list[float] | None = None
    ) -> float:
        """
        Calculates a weighted average, defaulting to an equal weighting if none provided.

        Args:
            scores: A list of scores to be averaged.
            weights: An optional list of weights. If None, equal weights are assumed.

        Returns:
            The calculated weighted average.

        Raises:
            WeightValidationError: If the weights are invalid (e.g., mismatched length).
        """
        if not scores:
            return ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L665_19", 0.0)

        if weights is None:
            # Equal weights
            weights = [ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L669_23", 1.0) / len(sco
res)] * len(scores)

        # Validate weights length matches scores length
        if len(weights) != len(scores):
            msg = (
                f"Weight length mismatch: {len(weights)} weights for {len(scores)} scores"
            )
            logger.error(msg)
            raise WeightValidationError(msg)

        # Validate weights sum to ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L679_34", 1.0
)
        valid, msg = self.validate_weights(weights)
        if not valid:
            # If validation failed and abort_on_insufficient is False,
            # validate_weights already logged the error and returned False
            # We should raise here to avoid silent failure
            raise WeightValidationError(msg)

        # Calculate weighted sum
        weighted_sum = sum(s * w for s, w in zip(scores, weights, strict=False))

        logger.debug(
            f"Weighted average calculated: "
            f"scores={scores}, weights={weights}, result={weighted_sum:.4f}"
        )

```

```
781:         return weighted_sum
782:
783:     def aggregate_with_sota(
784:         self,
785:         scores: list[float],
786:         weights: list[float] | None = None,
787:         method: str = "choquet",
788:         compute_uncertainty: bool = True,
789:     ) -> tuple[float, UncertaintyMetrics | None]:
790:         """
791:         SOTA aggregation with Choquet integral and uncertainty quantification.
792:
793:         This method provides:
794:         1. Non-linear aggregation via Choquet integral (captures synergies)
795:         2. Bayesian uncertainty quantification via bootstrap
796:         3. Full reproducibility with fixed random seed
797:
798:         Args:
799:             scores: Input scores to aggregate
800:             weights: Optional weights (default: uniform)
801:             method: Aggregation method ("choquet" or "weighted_average")
802:             compute_uncertainty: Whether to compute uncertainty metrics
803:
804:         Returns:
805:             Tuple of (aggregated_score, uncertainty_metrics)
806:             If compute_uncertainty=False, uncertainty_metrics is None
807:
808:         Raises:
809:             ValueError: If scores is empty or method invalid
810:         """
811:         if not scores:
812:             raise ValueError("Cannot aggregate empty score list")
813:
814:         if method == "choquet":
815:             # Use Choquet integral for non-linear aggregation
816:             choquet_adapter = create_default_choquet_adapter(len(scores))
817:             score = choquet_adapter.aggregate(scores, weights)
818:             logger.info(f"Choquet aggregation: {len(scores)} inputs â\206\222 {score:.4f}")
819:         elif method == "weighted_average":
820:             # Fall back to standard weighted average
821:             score = self.calculate_weighted_average(scores, weights)
822:         else:
823:             raise ValueError(f"Unknown aggregation method: {method}")
824:
825:         # Compute uncertainty if requested
826:         uncertainty = None
827:         if compute_uncertainty and self.bootstrap_aggregator:
828:             _, uncertainty = aggregate_with_uncertainty(
829:                 scores, weights, n_bootstrap=1000, random_seed=42
830:             )
831:             logger.debug(
832:                 f"Uncertainty: mean={uncertainty.mean:.4f}, "
833:                 f"std={uncertainty.std:.4f}, "
834:                 f"CI95={uncertainty.confidence_interval_95}"
835:             )
836:
```

```
  837:         return score, uncertainty
  838:
  839:     def apply_rubric_thresholds(
  840:         self,
  841:         score: float,
  842:         thresholds: dict[str, float] | None = None
  843:     ) -> str:
  844:         """
  845:         Apply rubric thresholds to determine quality level.
  846:
  847:         Args:
  848:             score: Aggregated score (0-3 range)
  849:             thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO, ACEPTABLE)
  850:                         Each value should be a normalized threshold (0-1 range)
  851:
  852:         Returns:
  853:             Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
  854:         """
  855:         # Clamp score to valid range [0, 3]
  856:         clamped_score = max(ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L714_28", 0.0), min
(3.0, score))
  857:
  858:         # Normalize to 0-1 range
  859:         normalized_score = clamped_score / 3.0
  860:
  861:         # Use provided thresholds or defaults
  862:         if thresholds:
  863:             excellent_threshold = thresholds.get('EXCELENTE', ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights
", "auto_param_L721_62", 0.85))
  864:             good_threshold = thresholds.get('BUENO', ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_
param_L722_53", 0.70))
  865:             acceptable_threshold = thresholds.get('ACEPTABLE', ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weight
s", "auto_param_L723_63", 0.55))
  866:         else:
  867:             excellent_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "excellent_threshold", 0.
85) # Refactored
  868:             good_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "good_threshold", 0.7) # Refac
tored
  869:             acceptable_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "acceptable_threshold",
0.55) # Refactored
  870:
  871:         # Apply thresholds
  872:         if normalized_score >= excellent_threshold:
  873:             quality = "EXCELENTE"
  874:         elif normalized_score >= good_threshold:
  875:             quality = "BUENO"
  876:         elif normalized_score >= acceptable_threshold:
  877:             quality = "ACEPTABLE"
  878:         else:
  879:             quality = "INSUFICIENTE"
  880:
  881:         logger.debug(
  882:             f"Rubric applied: score={score:.4f}, "
  883:             f"normalized={normalized_score:.4f}, quality={quality}"
  884:         )
  885:
```

```
886:         return quality
887:
888:     def aggregate_dimension(
889:         self,
890:         scored_results: list[ScoredResult],
891:         group_by_values: dict[str, Any],
892:         weights: list[float] | None = None,
893:     ) -> DimensionScore:
894:         """
895:         Aggregate a single dimension from micro question results.
896:
897:         Args:
898:             scored_results: List of scored results for this dimension/area.
899:             group_by_values: Dictionary of grouping keys and their values.
900:             weights: Optional weights for questions (defaults to equal weights).
901:
902:         Returns:
903:             DimensionScore with aggregated score and quality level.
904:
905:         Raises:
906:             ValidationError: If validation fails.
907:             CoverageError: If coverage is insufficient.
908:         """
909:         dimension_id = group_by_values.get("dimension", "UNKNOWN")
910:         area_id = group_by_values.get("policy_area", "UNKNOWN")
911:         logger.info(f"Aggregating dimension {dimension_id} for area {area_id}")
912:
913:         validation_details = {}
914:
915:         # In this context, scored_results are already grouped, so we can use them directly.
916:         dim_results = scored_results
917:
918:         expected_count = self._expected_question_count(area_id, dimension_id)
919:
920:         # Validate coverage
921:         try:
922:             coverage_valid, coverage_msg = self.validate_coverage(
923:                 dim_results,
924:                 expected_count=expected_count or 5,
925:             )
926:             validation_details["coverage"] = {
927:                 "valid": coverage_valid,
928:                 "message": coverage_msg,
929:                 "count": len(dim_results)
930:             }
931:         except CoverageError as e:
932:             logger.error(f"Coverage validation failed for {dimension_id}/{area_id}: {e}")
933:             # Return minimal score if aborted
934:             return DimensionScore(
935:                 dimension_id=dimension_id,
936:                 area_id=area_id,
937:                 score=ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L795_22", 0.0),
938:                 quality_level="INSUFICIENTE",
939:                 contributing_questions=[],
940:                 validation_passed=False,
941:                 validation_details={"error": str(e), "type": "coverage"}
```

```
            )

        if not dim_results:
            logger.warning(f"No results for dimension {dimension_id}/{area_id}")
            return DimensionScore(
                dimension_id=dimension_id,
                area_id=area_id,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.DimensionAggregator.validate_weights", "auto_param_L807_22", 0.0),
                quality_level="INSUFICIENTE",
                contributing_questions=[],
                validation_passed=False,
                validation_details={"error": "No results", "type": "empty"}
            )

        # Extract scores
        scores = [r.score for r in dim_results]
        question_ids = [r.question_global for r in dim_results]

        # Calculate weighted average with SOTA features
        resolved_weights = weights or self._resolve_dimension_weights(dimension_id, dim_results)

        # SOTA: Use Choquet + uncertainty if enabled
        if self.enable_sota_features and len(scores) >= 3:
            try:
                avg_score, uncertainty = self.aggregate_with_sota(
                    scores,
                    resolved_weights,
                    method="choquet",
                    compute_uncertainty=True,
                )
                validation_details["aggregation"] = {
                    "method": "choquet",
                    "uncertainty": uncertainty.to_dict() if uncertainty else None,
                }
            except Exception as e:
                logger.warning(f"SOTA aggregation failed, falling back to standard: {e}")
                avg_score = self.calculate_weighted_average(scores, resolved_weights)
                uncertainty = None
                validation_details["aggregation"] = {"method": "weighted_average", "fallback": True}
        else:
            # Standard aggregation
            avg_score = self.calculate_weighted_average(scores, resolved_weights)
            uncertainty = None
            validation_details["aggregation"] = {"method": "weighted_average"}

        validation_details["weights"] = {
            "valid": True,
            "weights": resolved_weights if resolved_weights else "equal",
            "score": avg_score
        }

        # Apply rubric thresholds
        quality_level = self.apply_rubric_thresholds(avg_score)
        validation_details["rubric"] = {
            "score": avg_score,
            "quality_level": quality_level
```

```
        }
        validation_details["score_max"] = 3.0

        # SOTA: Add provenance tracking
        provenance_node_id = f"DIM_{dimension_id}_{area_id}"
        if self.enable_sota_features and self.provenance_dag:
            # Add dimension node
            dim_node = ProvenanceNode(
                node_id=provenance_node_id,
                level="dimension",
                score=avg_score,
                quality_level=quality_level,
                metadata={
                    "dimension_id": dimension_id,
                    "area_id": area_id,
                    "n_questions": len(question_ids),
                },
            )
            self.provenance_dag.add_node(dim_node)

            # Add aggregation edges from questions to dimension
            question_node_ids = [f"Q{qid:03d}" for qid in question_ids]
            for qid_str, qid in zip(question_node_ids, question_ids):
                # Add question node if not exists
                if qid_str not in self.provenance_dag.nodes:
                    q_node = ProvenanceNode(
                        node_id=qid_str,
                        level="micro",
                        score=scores[question_ids.index(qid)],
                        quality_level="UNKNOWN",
                    )
                    self.provenance_dag.add_node(q_node)

            # Record aggregation operation
            self.provenance_dag.add_aggregation_edge(
                source_ids=question_node_ids,
                target_id=provenance_node_id,
                operation="choquet" if self.enable_sota_features else "weighted_average",
                weights=resolved_weights or [1.0 / len(question_ids)] * len(question_ids),
                metadata={"dimension": dimension_id, "area": area_id},
            )

            logger.debug(f"Provenance recorded: {len(question_node_ids)} questions â\206\222 {provenance_node_id}")

        logger.info(
            f"â\234\223 Dimension {dimension_id}/{area_id}: "
            f"score={avg_score:.4f}, quality={quality_level}"
            + (f", std={uncertainty.std:.4f}" if uncertainty else "")
        )

        return DimensionScore(
            dimension_id=dimension_id,
            area_id=area_id,
            score=avg_score,
            quality_level=quality_level,
            contributing_questions=question_ids,
```

```
            validation_passed=True,
            validation_details=validation_details,
            # SOTA fields
            score_std=uncertainty.std if uncertainty else 0.0,
            confidence_interval_95=uncertainty.confidence_interval_95 if uncertainty else (0.0, 0.0),
            epistemic_uncertainty=uncertainty.epistemic_uncertainty if uncertainty else 0.0,
            aleatoric_uncertainty=uncertainty.aleatoric_uncertainty if uncertainty else 0.0,
            provenance_node_id=provenance_node_id if self.enable_sota_features else "",
            aggregation_method="choquet" if (self.enable_sota_features and len(scores) >= 3) else "weighted_average",
        )

    def run(
        self,
        scored_results: list[ScoredResult],
        group_by_keys: list[str]
    ) -> list[DimensionScore]:
        """
        Run the dimension aggregation process.

        Args:
            scored_results: List of all scored results.
            group_by_keys: List of keys to group by.

        Returns:
            A list of DimensionScore objects.
        """
        def key_func(r):
            return tuple(getattr(r, key) for key in group_by_keys)
        grouped_results = group_by(scored_results, key_func)

        dimension_scores = []
        for group_key, results in grouped_results.items():
            group_by_values = dict(zip(group_by_keys, group_key, strict=False))
            score = self.aggregate_dimension(results, group_by_values)
            dimension_scores.append(score)

        return dimension_scores

    @calibrated_method("farfan_core.processing.aggregation.DimensionAggregator._expected_question_count")
    def _expected_question_count(self, area_id: str, dimension_id: str) -> int | None:
        if not self.aggregation_settings.dimension_expected_counts:
            return None
        return self.aggregation_settings.dimension_expected_counts.get((area_id, dimension_id))

    def _resolve_dimension_weights(
        self,
        dimension_id: str,
        dim_results: list[ScoredResult],
    ) -> list[float] | None:
        mapping = self.aggregation_settings.dimension_question_weights.get(dimension_id)
        if not mapping:
            return None

        weights: list[float] = []
        for result in dim_results:
            slot = result.base_slot
```

```
1110:             weight = mapping.get(slot)
1111:             if weight is None:
1112:                 logger.debug(
1113:                     "Missing weight for slot %s in dimension %s â\200\223 falling back to equal weights",
1114:                     slot,
1115:                     dimension_id,
1116:                 )
1117:                 return None
1118:             weights.append(weight)
1119:
1120:         total = sum(weights)
1121:         if total <= 0:
1122:             return None
1123:         return [w / total for w in weights]
1124:
1125: def run_aggregation_pipeline(
1126:     scored_results: list[dict[str, Any]],
1127:     monolith: dict[str, Any],
1128:     abort_on_insufficient: bool = True
1129: ) -> list[ClusterScore]:
1130:     """
1131:     Orchestrates the end-to-end aggregation pipeline.
1132:
1133:     This function provides a high-level entry point to the aggregation system,
1134:     demonstrating the sequential wiring of the aggregator components. It ensures
1135:     that data flows from raw scored results through dimension, area, and
1136:     finally cluster aggregation in a controlled and validated manner.
1137:
1138:     Note on Parallelization: This implementation is sequential. For very large
1139:     datasets, the `group_by` operations in each aggregator's `run` method
1140:     could be parallelized (e.g., using `concurrent.futures`) to process
1141:     independent groups concurrently.
1142:
1143:     Args:
1144:         scored_results: A list of dictionaries, each representing a raw scored result.
1145:         monolith: The central monolith configuration object.
1146:         abort_on_insufficient: If True, the pipeline will stop on validation errors.
1147:
1148:     Returns:
1149:         A list of aggregated ClusterScore objects.
1150:     """
1151:     # 1. Input Validation (Pre-flight check)
1152:     validated_scored_results = validate_scored_results(scored_results)
1153:
1154:     aggregation_settings = AggregationSettings.from_monolith(monolith)
1155:
1156:     # 2. FASE 4: Dimension Aggregation
1157:     dim_aggregator = DimensionAggregator(
1158:         monolith,
1159:         abort_on_insufficient,
1160:         aggregation_settings=aggregation_settings,
1161:     )
1162:     dimension_scores = dim_aggregator.run(
1163:         validated_scored_results,
1164:         group_by_keys=dim_aggregator.dimension_group_by_keys,
1165:     )
```

```
1166:
1167:     # 3. FASE 5: Area Policy Aggregation
1168:     area_aggregator = AreaPolicyAggregator(
1169:         monolith,
1170:         abort_on_insufficient,
1171:         aggregation_settings=aggregation_settings,
1172:     )
1173:     area_scores = area_aggregator.run(
1174:         dimension_scores,
1175:         group_by_keys=area_aggregator.area_group_by_keys,
1176:     )
1177:
1178:     # 4. FASE 6: Cluster Aggregation
1179:     cluster_aggregator = ClusterAggregator(
1180:         monolith,
1181:         abort_on_insufficient,
1182:         aggregation_settings=aggregation_settings,
1183:     )
1184:     cluster_definitions = monolith["blocks"]["niveles_abstraccion"]["clusters"]
1185:     cluster_scores = cluster_aggregator.run(
1186:         area_scores,
1187:         cluster_definitions
1188:     )
1189:
1190:     return cluster_scores
1191:
1192:     def run(
1193:         self,
1194:         scored_results: list[ScoredResult],
1195:         group_by_keys: list[str]
1196:     ) -> list[DimensionScore]:
1197:         """
1198:         Run the dimension aggregation process.
1199:
1200:         Args:
1201:             scored_results: List of all scored results.
1202:             group_by_keys: List of keys to group by.
1203:
1204:         Returns:
1205:             A list of DimensionScore objects.
1206:         """
1207:         def key_func(r):
1208:             return tuple(getattr(r, key) for key in group_by_keys)
1209:         grouped_results = group_by(scored_results, key_func)
1210:
1211:         dimension_scores = []
1212:         for group_key, results in grouped_results.items():
1213:             group_by_values = dict(zip(group_by_keys, group_key, strict=False))
1214:             score = self.aggregate_dimension(results, group_by_values)
1215:             dimension_scores.append(score)
1216:
1217:         return dimension_scores
1218:
1219: class AreaPolicyAggregator:
1220:     """
1221:     Aggregates dimension scores into policy area scores.
```

```
1222:
1223:     Responsibilities:
1224:     - Aggregate 6 dimension scores per policy area
1225:     - Validate dimension completeness
1226:     - Apply area-level rubric thresholds
1227:     - Ensure hermeticity (no dimension overlap)
1228:     """
1229:
1230:     def __init__(
1231:         self,
1232:         monolith: dict[str, Any] | None = None,
1233:         abort_on_insufficient: bool = True,
1234:         aggregation_settings: AggregationSettings | None = None,
1235:     ) -> None:
1236:         """
1237:         Initialize area aggregator.
1238:
1239:         Args:
1240:             monolith: Questionnaire monolith configuration (optional, required for run())
1241:             abort_on_insufficient: Whether to abort on insufficient coverage
1242:
1243:         Raises:
1244:             ValueError: If monolith is None and required for operations
1245:         """
1246:         self.monolith = monolith
1247:         self.abort_on_insufficient = abort_on_insufficient
1248:         self.aggregation_settings = aggregation_settings or AggregationSettings.from_monolith(monolith)
1249:         self.area_group_by_keys = self.aggregation_settings.area_group_by_keys or ["area_id"]
1250:
1251:         # Extract configuration if monolith provided
1252:         if monolith is not None:
1253:             self.scoring_config = monolith["blocks"]["scoring"]
1254:             self.niveles = monolith["blocks"]["niveles_abstraccion"]
1255:             self.policy_areas = self.niveles["policy_areas"]
1256:             self.dimensions = self.niveles["dimensions"]
1257:         else:
1258:             self.scoring_config = None
1259:             self.niveles = None
1260:             self.policy_areas = None
1261:             self.dimensions = None
1262:
1263:         logger.info("AreaPolicyAggregator initialized")
1264:
1265:     def validate_hermeticity(
1266:         self,
1267:         dimension_scores: list[DimensionScore],
1268:         area_id: str
1269:     ) -> tuple[bool, str]:
1270:         """
1271:         Validate hermeticity (no dimension overlap/gaps).
1272:         Uses scoped validation based on policy_area.dimension_ids from monolith.
1273:
1274:         Args:
1275:             dimension_scores: List of dimension scores for the area
1276:             area_id: Policy area ID
1277:
```

```
        Returns:
            Tuple of (is_valid, message)

        Raises:
            HermeticityValidationError: If hermeticity is violated
        """
        # Get expected dimensions for this specific policy area
        area_def = next(
            (a for a in self.policy_areas if a["policy_area_id"] == area_id),
            None
        )

        if area_def and "dimension_ids" in area_def:
            expected_dimension_ids = set(area_def["dimension_ids"])
        else:
            # Fallback to all global dimensions if not specified
            expected_dimension_ids = {d["dimension_id"] for d in self.dimensions}

        actual_dimension_ids = {d.dimension_id for d in dimension_scores}
        len(expected_dimension_ids)
        len(dimension_scores)

        # Check for missing dimensions
        missing_dims = expected_dimension_ids - actual_dimension_ids
        if missing_dims:
            msg = (
                f"Hermeticity violation for area {area_id}: "
                f"missing dimensions {missing_dims}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        # Check for unexpected dimensions
        extra_dims = actual_dimension_ids - expected_dimension_ids
        if extra_dims:
            msg = (
                f"Hermeticity violation for area {area_id}: "
                f"unexpected dimensions {extra_dims}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        # Check for duplicate dimensions
        dimension_ids = [d.dimension_id for d in dimension_scores]
        if len(dimension_ids) != len(set(dimension_ids)):
            msg = f"Hermeticity violation for area {area_id}: duplicate dimensions found"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        logger.debug(f"Hermeticity validation passed for area {area_id}")
```

```
1334:         return True, "Hermeticity validated"
1335:
1336:     @calibrated_method("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores")
1337:     def normalize_scores(self, dimension_scores: list[DimensionScore]) -> list[float]:
1338:         """
1339:         Normalize dimension scores to 0-1 range.
1340:
1341:         Args:
1342:             dimension_scores: List of dimension scores
1343:
1344:         Returns:
1345:             List of normalized scores
1346:         """
1347:         normalized = []
1348:         for d in dimension_scores:
1349:             # Extract max_expected from validation_details or default to 3.0
1350:             max_expected = d.validation_details.get('score_max', 3.0) if d.validation_details else 3.0
1351:             normalized.append(max(ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1148_34", 0
.0), min(max_expected, d.score)) / max_expected)
1352:
1353:         logger.debug(f"Scores normalized: {normalized}")
1354:         return normalized
1355:
1356:     def apply_rubric_thresholds(
1357:         self,
1358:         score: float,
1359:         thresholds: dict[str, float] | None = None
1360:     ) -> str:
1361:         """
1362:         Apply area-level rubric thresholds.
1363:
1364:         Args:
1365:             score: Aggregated score (0-3 range)
1366:             thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO, ACEPTABLE)
1367:                         Each value should be a normalized threshold (0-1 range)
1368:
1369:         Returns:
1370:             Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
1371:         """
1372:         # Clamp score to valid range [0, 3]
1373:         clamped_score = max(ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1170_28", 0.0), m
in(3.0, score))
1374:
1375:         # Normalize to 0-1 range
1376:         normalized_score = clamped_score / 3.0
1377:
1378:         # Use provided thresholds or defaults
1379:         if thresholds:
1380:             excellent_threshold = thresholds.get('EXCELENTE', ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_score
s", "auto_param_L1177_62", 0.85))
1381:             good_threshold = thresholds.get('BUENO', ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto
_param_L1178_53", 0.70))
1382:             acceptable_threshold = thresholds.get('ACEPTABLE', ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scor
es", "auto_param_L1179_63", 0.55))
1383:         else:
1384:             excellent_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "excellent_threshold", 0
```

```
.85) # Refactored
 1385:             good_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "good_threshold", 0.7) # Refa
ctored
 1386:             acceptable_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "acceptable_threshold",
 0.55) # Refactored
 1387:
 1388:         # Apply thresholds
 1389:         if normalized_score >= excellent_threshold:
 1390:             quality = "EXCELENTE"
 1391:         elif normalized_score >= good_threshold:
 1392:             quality = "BUENO"
 1393:         elif normalized_score >= acceptable_threshold:
 1394:             quality = "ACEPTABLE"
 1395:         else:
 1396:             quality = "INSUFICIENTE"
 1397:
 1398:         logger.debug(
 1399:             f"Area rubric applied: score={score:.4f}, "
 1400:             f"normalized={normalized_score:.4f}, quality={quality}"
 1401:         )
 1402:
 1403:         return quality
 1404:
 1405:     def aggregate_area(
 1406:         self,
 1407:         dimension_scores: list[DimensionScore],
 1408:         group_by_values: dict[str, Any],
 1409:         weights: list[float] | None = None,
 1410:     ) -> AreaScore:
 1411:         """
 1412:         Aggregate a single policy area from dimension scores.
 1413:
 1414:         Args:
 1415:             dimension_scores: List of dimension scores for this area.
 1416:             group_by_values: Dictionary of grouping keys and their values.
 1417:             weights: Optional list of weights for dimension scores.
 1418:
 1419:         Returns:
 1420:             AreaScore with aggregated score and quality level.
 1421:
 1422:         Raises:
 1423:             ValidationError: If validation fails.
 1424:         """
 1425:         area_id = group_by_values.get("area_id", "UNKNOWN")
 1426:         logger.info(f"Aggregating policy area {area_id}")
 1427:
 1428:         validation_details = {}
 1429:
 1430:         # The dimension_scores are already grouped.
 1431:         area_dim_scores = dimension_scores
 1432:
 1433:         # Validate hermeticity
 1434:         try:
 1435:             hermetic_valid, hermetic_msg = self.validate_hermeticity(area_dim_scores, area_id)
 1436:             validation_details["hermeticity"] = {
 1437:                 "valid": hermetic_valid,
```

```
                "message": hermetic_msg,
                "dimension_count": len(area_dim_scores)
            }
        except HermeticityValidationError as e:
            logger.error(f"Hermeticity validation failed for area {area_id}: {e}")
            # Get area name
            area_name = next(
                (a["i18n"]["keys"]["label_es"] for a in self.policy_areas
                 if a["policy_area_id"] == area_id),
                area_id
            )
            return AreaScore(
                area_id=area_id,
                area_name=area_name,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1249_22", 0.0),
                quality_level="INSUFICIENTE",
                dimension_scores=[],
                validation_passed=False,
                validation_details={"error": str(e), "type": "hermeticity"}
            )

        if not area_dim_scores:
            logger.warning(f"No dimension scores for area {area_id}")
            area_name = next(
                (a["i18n"]["keys"]["label_es"] for a in self.policy_areas
                 if a["policy_area_id"] == area_id),
                area_id
            )
            return AreaScore(
                area_id=area_id,
                area_name=area_name,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1266_22", 0.0),
                quality_level="INSUFICIENTE",
                dimension_scores=[],
                validation_passed=False,
                validation_details={"error": "No dimensions", "type": "empty"}
            )

        # Normalize scores
        normalized = self.normalize_scores(area_dim_scores)
        validation_details["normalization"] = {
            "original": [d.score for d in area_dim_scores],
            "normalized": normalized
        }

        # Calculate weighted average score
        scores = [d.score for d in area_dim_scores]
        resolved_weights = weights or self._resolve_area_weights(area_id, area_dim_scores)
        avg_score = DimensionAggregator().calculate_weighted_average(scores, weights=resolved_weights)

        # Apply rubric thresholds
        quality_level = self.apply_rubric_thresholds(avg_score)
        validation_details["rubric"] = {
            "score": avg_score,
            "quality_level": quality_level
        }
```

```
1494:
1495:         # Get area name
1496:         area_name = next(
1497:             (a["i18n"]["keys"]["label_es"] for a in self.policy_areas
1498:              if a["policy_area_id"] == area_id),
1499:             area_id
1500:         )
1501:
1502:         logger.info(
1503:             f"â\234\223 Policy area {area_id} ({area_name}): "
1504:             f"score={avg_score:.4f}, quality={quality_level}"
1505:         )
1506:
1507:         return AreaScore(
1508:             area_id=area_id,
1509:             area_name=area_name,
1510:             score=avg_score,
1511:             quality_level=quality_level,
1512:             dimension_scores=area_dim_scores,
1513:             validation_passed=True,
1514:             validation_details=validation_details
1515:         )
1516:
1517:     def run(
1518:         self,
1519:         dimension_scores: list[DimensionScore],
1520:         group_by_keys: list[str]
1521:     ) -> list[AreaScore]:
1522:         """
1523:         Run the area aggregation process.
1524:
1525:         Args:
1526:             dimension_scores: List of all dimension scores.
1527:             group_by_keys: List of keys to group by.
1528:
1529:         Returns:
1530:             A list of AreaScore objects.
1531:         """
1532:         def key_func(d):
1533:             return tuple(getattr(d, key) for key in group_by_keys)
1534:         grouped_scores = group_by(dimension_scores, key_func)
1535:
1536:         area_scores = []
1537:         for group_key, scores in grouped_scores.items():
1538:             group_by_values = dict(zip(group_by_keys, group_key, strict=False))
1539:             score = self.aggregate_area(scores, group_by_values, weights=None)
1540:             area_scores.append(score)
1541:
1542:         return area_scores
1543:
1544:     def _resolve_area_weights(
1545:         self,
1546:         area_id: str,
1547:         dimension_scores: list[DimensionScore],
1548:     ) -> list[float] | None:
1549:         mapping = self.aggregation_settings.policy_area_dimension_weights.get(area_id)
```

```
        if not mapping:
            return None

        weights: list[float] = []
        for dim_score in dimension_scores:
            weight = mapping.get(dim_score.dimension_id)
            if weight is None:
                logger.debug(
                    "Missing weight for dimension %s in area %s â\200\223 falling back to equal weights",
                    dim_score.dimension_id,
                    area_id,
                )
                return None
            weights.append(weight)

        total = sum(weights)
        if total <= 0:
            return None
        return [w / total for w in weights]

class ClusterAggregator:
    """
    Aggregates policy area scores into cluster scores (MESO level).

    Responsibilities:
    - Aggregate multiple area scores per cluster
    - Apply cluster-specific weights
    - Calculate coherence metrics
    - Validate cluster hermeticity
    """

    PENALTY_WEIGHT = ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "PENALTY_WEIGHT", 0.3)
    MAX_SCORE = 3.0

    def __init__(
        self,
        monolith: dict[str, Any] | None = None,
        abort_on_insufficient: bool = True,
        aggregation_settings: AggregationSettings | None = None,
    ) -> None:
        """
        Initialize cluster aggregator.

        Args:
            monolith: Questionnaire monolith configuration (optional, required for run())
            abort_on_insufficient: Whether to abort on insufficient coverage

        Raises:
            ValueError: If monolith is None and required for operations
        """
        self.monolith = monolith
        self.abort_on_insufficient = abort_on_insufficient
        self.aggregation_settings = aggregation_settings or AggregationSettings.from_monolith(monolith)
        self.cluster_group_by_keys = self.aggregation_settings.cluster_group_by_keys or ["cluster_id"]

        # Extract configuration if monolith provided
```

```
1606:         if monolith is not None:
1607:             self.scoring_config = monolith["blocks"]["scoring"]
1608:             self.niveles = monolith["blocks"]["niveles_abstraccion"]
1609:             self.clusters = self.niveles["clusters"]
1610:         else:
1611:             self.scoring_config = None
1612:             self.niveles = None
1613:             self.clusters = None
1614:
1615:         logger.info("ClusterAggregator initialized")
1616:
1617:     def validate_cluster_hermeticity(
1618:         self,
1619:         cluster_def: dict[str, Any],
1620:         area_scores: list[AreaScore]
1621:     ) -> tuple[bool, str]:
1622:         """
1623:         Validate cluster hermeticity.
1624:
1625:         Args:
1626:             cluster_def: Cluster definition from monolith
1627:             area_scores: List of area scores for this cluster
1628:
1629:         Returns:
1630:             Tuple of (is_valid, message)
1631:
1632:         Raises:
1633:             HermeticityValidationError: If hermeticity is violated
1634:         """
1635:         expected_areas = cluster_def.get("policy_area_ids", [])
1636:         actual_areas = [a.area_id for a in area_scores]
1637:
1638:         # Check for duplicate areas
1639:         if len(actual_areas) != len(set(actual_areas)):
1640:             msg = (
1641:                 f"Cluster hermeticity violation: "
1642:                 f"duplicate areas found for cluster {cluster_def['cluster_id']}"
1643:             )
1644:             logger.error(msg)
1645:             if self.abort_on_insufficient:
1646:                 raise HermeticityValidationError(msg)
1647:             return False, msg
1648:
1649:         # Check that all expected areas are present
1650:         missing_areas = set(expected_areas) - set(actual_areas)
1651:         if missing_areas:
1652:             msg = (
1653:                 f"Cluster hermeticity violation: "
1654:                 f"missing areas {missing_areas} for cluster {cluster_def['cluster_id']}"
1655:             )
1656:             logger.error(msg)
1657:             if self.abort_on_insufficient:
1658:                 raise HermeticityValidationError(msg)
1659:             return False, msg
1660:
1661:         # Check for unexpected areas
```

```
1662:         extra_areas = set(actual_areas) - set(expected_areas)
1663:         if extra_areas:
1664:             msg = (
1665:                 f"Cluster hermeticity violation: "
1666:                 f"unexpected areas {extra_areas} for cluster {cluster_def['cluster_id']}"
1667:             )
1668:             logger.error(msg)
1669:             if self.abort_on_insufficient:
1670:                 raise HermeticityValidationError(msg)
1671:             return False, msg
1672:
1673:         logger.debug(f"Cluster hermeticity validated for {cluster_def['cluster_id']}")
1674:         return True, "Cluster hermeticity validated"
1675:
1676:     def apply_cluster_weights(
1677:         self,
1678:         area_scores: list[AreaScore],
1679:         weights: list[float] | None = None
1680:     ) -> float:
1681:         """
1682:         Apply cluster-specific weights to area scores.
1683:
1684:         Args:
1685:             area_scores: List of area scores
1686:             weights: Optional weights (defaults to equal weights)
1687:
1688:         Returns:
1689:             Weighted average score
1690:
1691:         Raises:
1692:             WeightValidationError: If weights validation fails
1693:         """
1694:         scores = [a.score for a in area_scores]
1695:
1696:         if weights is None:
1697:             # Equal weights
1698:             weights = [ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1495_23", 1.0) / len(s
cores)] * len(scores)
1699:
1700:         # Validate weights length matches scores length
1701:         if len(weights) != len(scores):
1702:             msg = (
1703:                 f"Cluster weight length mismatch: "
1704:                 f"{len(weights)} weights for {len(scores)} area scores"
1705:             )
1706:             logger.error(msg)
1707:             if self.abort_on_insufficient:
1708:                 raise WeightValidationError(msg)
1709:
1710:         # Validate weights sum to ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1507_34", 1
.0)
1711:         weight_sum = sum(weights)
1712:         tolerance = 1e-6
1713:         if abs(weight_sum - ParameterLoaderV2.get("farfan_core.processing.aggregation.AreaPolicyAggregator.normalize_scores", "auto_param_L1510_28", 1.0)) >
 tolerance:
1714:             msg = f"Cluster weight validation failed: sum={weight_sum:.6f}"
```

```
1715:             logger.error(msg)
1716:             if self.abort_on_insufficient:
1717:                 raise WeightValidationError(msg)
1718:
1719:         # Calculate weighted average
1720:         weighted_avg = sum(s * w for s, w in zip(scores, weights, strict=False))
1721:
1722:         logger.debug(
1723:             f"Cluster weights applied: scores={scores}, "
1724:             f"weights={weights}, result={weighted_avg:.4f}"
1725:         )
1726:
1727:         return weighted_avg
1728:
1729:     @calibrated_method("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence")
1730:     def analyze_coherence(self, area_scores: list[AreaScore]) -> float:
1731:         """
1732:         Analyze cluster coherence.
1733:
1734:         Coherence is measured as the inverse of standard deviation.
1735:         Higher coherence means scores are more consistent.
1736:
1737:         Args:
1738:             area_scores: List of area scores
1739:
1740:         Returns:
1741:             Coherence value (0-1, where 1 is perfect coherence)
1742:         """
1743:         scores = [a.score for a in area_scores]
1744:
1745:         if len(scores) <= 1:
1746:             return ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1543_19", 1.0)
1747:
1748:         # Calculate mean
1749:         mean = sum(scores) / len(scores)
1750:
1751:         # Calculate standard deviation
1752:         variance = sum((s - mean) ** 2 for s in scores) / len(scores)
1753:         std_dev = variance ** ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1550_30", 0.5)
1754:
1755:         # Convert to coherence (inverse relationship)
1756:         # Normalize by max possible std dev (3.0 for 0-3 range)
1757:         max_std = 3.0
1758:         coherence = max(ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1555_24", 0.0), Paramet
erLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1555_29", 1.0) - (std_dev / max_std))
1759:
1760:         logger.debug(
1761:             f"Coherence analysis: mean={mean:.4f}, "
1762:             f"std_dev={std_dev:.4f}, coherence={coherence:.4f}"
1763:         )
1764:
1765:         return coherence
1766:
1767:     def aggregate_cluster(
1768:         self,
1769:         area_scores: list[AreaScore],
```

```
        group_by_values: dict[str, Any],
        weights: list[float] | None = None,
    ) -> ClusterScore:
        """
        Aggregate a single MESO cluster from area scores.

        Args:
            area_scores: List of area scores for this cluster.
            group_by_values: Dictionary of grouping keys and their values.
            weights: Optional cluster-specific weights.

        Returns:
            ClusterScore with aggregated score and coherence.

        Raises:
            ValidationError: If validation fails.
        """
        cluster_id = group_by_values.get("cluster_id", "UNKNOWN")
        logger.info(f"Aggregating cluster {cluster_id}")

        validation_details = {}

        # Get cluster definition
        cluster_def = next(
            (c for c in self.clusters if c["cluster_id"] == cluster_id), None
        )

        if not cluster_def:
            logger.error(f"Cluster definition not found: {cluster_id}")
            return ClusterScore(
                cluster_id=cluster_id,
                cluster_name=cluster_id,
                areas=[],
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1600_22", 0.0),
                coherence=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1601_26", 0.0),
                variance=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1602_25", 0.0),
                weakest_area=None,
                area_scores=[],
                validation_passed=False,
                validation_details={"error": "Definition not found", "type": "config"},
            )

        cluster_name = cluster_def["i18n"]["keys"]["label_es"]
        expected_areas = cluster_def["policy_area_ids"]

        # The area_scores are already grouped.
        cluster_area_scores = area_scores

        # Validate hermeticity
        try:
            hermetic_valid, hermetic_msg = self.validate_cluster_hermeticity(
                cluster_def,
                cluster_area_scores
            )
            validation_details["hermeticity"] = {
                "valid": hermetic_valid,
```

```
                "message": hermetic_msg
            }
        except HermeticityValidationError as e:
            logger.error(f"Cluster hermeticity validation failed: {e}")
            return ClusterScore(
                cluster_id=cluster_id,
                cluster_name=cluster_name,
                areas=expected_areas,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1631_22", 0.0),
                coherence=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1632_26", 0.0),
                variance=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1633_25", 0.0),
                weakest_area=None,
                area_scores=[],
                validation_passed=False,
                validation_details={"error": str(e), "type": "hermeticity"}
            )

        if not cluster_area_scores:
            logger.warning(f"No area scores for cluster {cluster_id}")
            return ClusterScore(
                cluster_id=cluster_id,
                cluster_name=cluster_name,
                areas=expected_areas,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1646_22", 0.0),
                coherence=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1647_26", 0.0),
                variance=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1648_25", 0.0),
                weakest_area=None,
                area_scores=[],
                validation_passed=False,
                validation_details={"error": "No areas", "type": "empty"}
            )

        # Apply cluster weights
        resolved_weights = weights or self._resolve_cluster_weights(cluster_id, cluster_area_scores)
        try:
            weighted_score = self.apply_cluster_weights(cluster_area_scores, resolved_weights)
            validation_details["weights"] = {
                "valid": True,
                "weights": resolved_weights if resolved_weights else "equal",
                "score": weighted_score
            }
        except WeightValidationError as e:
            logger.error(f"Cluster weight validation failed: {e}")
            return ClusterScore(
                cluster_id=cluster_id,
                cluster_name=cluster_name,
                areas=expected_areas,
                score=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1670_22", 0.0),
                coherence=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1671_26", 0.0),
                variance=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1672_25", 0.0),
                weakest_area=None,
                area_scores=cluster_area_scores,
                validation_passed=False,
                validation_details={"error": str(e), "type": "weights"}
            )

```

```
1882:         # Analyze coherence and variance metrics
1883:         coherence = self.analyze_coherence(cluster_area_scores)
1884:         scores_array = [a.score for a in cluster_area_scores]
1885:         if scores_array:
1886:             mean_score = sum(scores_array) / len(scores_array)
1887:             variance = sum((score - mean_score) ** 2 for score in scores_array) / len(scores_array)
1888:         else:
1889:             variance = ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "variance", 0.0) # Refactored
1890:         weakest_area = min(cluster_area_scores, key=lambda a: a.score, default=None)
1891:
1892:         std_dev = variance ** ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1689_30", 0.5)
1893:         normalized_std = min(std_dev / self.MAX_SCORE, ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto
_param_L1690_55", 1.0)) if std_dev > 0 else ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1690_80",
0.0)
1894:         penalty_factor = ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1691_25", 1.0) - (norm
alized_std * self.PENALTY_WEIGHT)
1895:         adjusted_score = weighted_score * penalty_factor
1896:
1897:         validation_details["coherence"] = {
1898:             "value": coherence,
1899:             "interpretation": "high" if coherence > ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_pa
ram_L1696_52", 0.8) else "medium" if coherence > ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1696_
85", 0.6) else "low"
1900:         }
1901:         validation_details["variance"] = variance
1902:         if weakest_area:
1903:             validation_details["weakest_area"] = weakest_area.area_id
1904:         validation_details["imbalance_penalty"] = {
1905:             "std_dev": std_dev,
1906:             "penalty_factor": penalty_factor,
1907:             "raw_score": weighted_score,
1908:             "adjusted_score": adjusted_score,
1909:         }
1910:
1911:         logger.info(
1912:             f"â\234\223 Cluster {cluster_id} ({cluster_name}): "
1913:             f"score={adjusted_score:.4f}, coherence={coherence:.4f}"
1914:         )
1915:
1916:         return ClusterScore(
1917:             cluster_id=cluster_id,
1918:             cluster_name=cluster_name,
1919:             areas=expected_areas,
1920:             score=adjusted_score,
1921:             coherence=coherence,
1922:             variance=variance,
1923:             weakest_area=weakest_area.area_id if weakest_area else None,
1924:             area_scores=cluster_area_scores,
1925:             validation_passed=True,
1926:             validation_details=validation_details
1927:         )
1928:
1929:     def run(
1930:         self,
1931:         area_scores: list[AreaScore],
1932:         cluster_definitions: list[dict[str, Any]]
```

```
    ) -> list[ClusterScore]:
        """
        Run the cluster aggregation process.

        Args:
            area_scores: List of all area scores.
            cluster_definitions: List of cluster definitions from the monolith.

        Returns:
            A list of ClusterScore objects.
        """
        # Create a mapping from area_id to cluster_id
        area_to_cluster = {}
        for cluster in cluster_definitions:
            for area_id in cluster["policy_area_ids"]:
                area_to_cluster[area_id] = cluster["cluster_id"]

        # Assign cluster_id to each area score
        for score in area_scores:
            score.cluster_id = area_to_cluster.get(score.area_id)

        def key_func(area_score: AreaScore) -> tuple:
            return tuple(getattr(area_score, key) for key in self.cluster_group_by_keys)

        grouped_scores = group_by([s for s in area_scores if hasattr(s, 'cluster_id')], key_func)

        cluster_scores = []
        for group_key, scores in grouped_scores.items():
            group_by_values = dict(zip(self.cluster_group_by_keys, group_key, strict=False))
            score = self.aggregate_cluster(scores, group_by_values)
            cluster_scores.append(score)

        return cluster_scores

    def _resolve_cluster_weights(
        self,
        cluster_id: str,
        area_scores: list[AreaScore],
    ) -> list[float] | None:
        mapping = self.aggregation_settings.cluster_policy_area_weights.get(cluster_id)
        if not mapping:
            return None

        weights: list[float] = []
        for area_score in area_scores:
            weight = mapping.get(area_score.area_id)
            if weight is None:
                logger.debug(
                    "Missing weight for area %s in cluster %s â\200\223 falling back to equal weights",
                    area_score.area_id,
                    cluster_id,
                )
                return None
            weights.append(weight)

        total = sum(weights)
```

```
1989:         if total <= 0:
1990:             return None
1991:         return [w / total for w in weights]
1992:
1993: class MacroAggregator:
1994:     """
1995:     Performs holistic macro evaluation (Q305).
1996:
1997:     Responsibilities:
1998:     - Aggregate all cluster scores
1999:     - Calculate cross-cutting coherence
2000:     - Identify systemic gaps
2001:     - Assess strategic alignment
2002:     """
2003:
2004:     def __init__(
2005:         self,
2006:         monolith: dict[str, Any] | None = None,
2007:         abort_on_insufficient: bool = True,
2008:         aggregation_settings: AggregationSettings | None = None,
2009:     ) -> None:
2010:         """
2011:         Initialize macro aggregator.
2012:
2013:         Args:
2014:             monolith: Questionnaire monolith configuration (optional, required for run())
2015:             abort_on_insufficient: Whether to abort on insufficient coverage
2016:
2017:         Raises:
2018:             ValueError: If monolith is None and required for operations
2019:         """
2020:         self.monolith = monolith
2021:         self.abort_on_insufficient = abort_on_insufficient
2022:         self.aggregation_settings = aggregation_settings or AggregationSettings.from_monolith(monolith)
2023:
2024:         # Extract configuration if monolith provided
2025:         if monolith is not None:
2026:             self.scoring_config = monolith["blocks"]["scoring"]
2027:             self.niveles = monolith["blocks"]["niveles_abstraccion"]
2028:         else:
2029:             self.scoring_config = None
2030:             self.niveles = None
2031:
2032:         logger.info("MacroAggregator initialized")
2033:
2034:     def calculate_cross_cutting_coherence(
2035:         self,
2036:         cluster_scores: list[ClusterScore]
2037:     ) -> float:
2038:         """
2039:         Calculate cross-cutting coherence across all clusters.
2040:
2041:         Args:
2042:             cluster_scores: List of cluster scores
2043:
2044:         Returns:
```

```
2045:             Cross-cutting coherence value (0-1)
2046:         """
2047:         scores = [c.score for c in cluster_scores]
2048:
2049:         if len(scores) <= 1:
2050:             return ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1847_19", 1.0)
2051:
2052:         # Calculate mean
2053:         mean = sum(scores) / len(scores)
2054:
2055:         # Calculate standard deviation
2056:         variance = sum((s - mean) ** 2 for s in scores) / len(scores)
2057:         std_dev = variance ** ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1854_30", 0.5)
2058:
2059:         # Convert to coherence
2060:         max_std = 3.0
2061:         coherence = max(ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1858_24", 0.0), Paramet
erLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1858_29", 1.0) - (std_dev / max_std))
2062:
2063:         logger.debug(
2064:             f"Cross-cutting coherence: mean={mean:.4f}, "
2065:             f"std_dev={std_dev:.4f}, coherence={coherence:.4f}"
2066:         )
2067:
2068:         return coherence
2069:
2070:     def identify_systemic_gaps(
2071:         self,
2072:         area_scores: list[AreaScore]
2073:     ) -> list[str]:
2074:         """
2075:         Identify systemic gaps (areas with INSUFICIENTE quality).
2076:
2077:         Args:
2078:             area_scores: List of area scores
2079:
2080:         Returns:
2081:             List of area names with systemic gaps
2082:         """
2083:         gaps = []
2084:         for area in area_scores:
2085:             if area.quality_level == "INSUFICIENTE":
2086:                 gaps.append(area.area_name)
2087:                 logger.warning(f"Systemic gap identified: {area.area_name}")
2088:
2089:         logger.info(f"Systemic gaps identified: {len(gaps)}")
2090:         return gaps
2091:
2092:     def assess_strategic_alignment(
2093:         self,
2094:         cluster_scores: list[ClusterScore],
2095:         dimension_scores: list[DimensionScore]
2096:     ) -> float:
2097:         """
2098:         Assess strategic alignment across all levels.
2099:
```

```
2100:         Args:
2101:             cluster_scores: List of cluster scores
2102:             dimension_scores: List of dimension scores
2103:
2104:         Returns:
2105:             Strategic alignment score (0-1)
2106:         """
2107:         # Calculate average cluster coherence
2108:         cluster_coherence = (
2109:             sum(c.coherence for c in cluster_scores) / len(cluster_scores)
2110:             if cluster_scores else ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1907_35", 0.
0)
2111:         )
2112:
2113:         # Calculate dimension validation rate
2114:         validated_dims = sum(1 for d in dimension_scores if d.validation_passed)
2115:         validation_rate = validated_dims / len(dimension_scores) if dimension_scores else ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterA
ggregator.analyze_coherence", "auto_param_L1912_90", 0.0)
2116:
2117:         # Strategic alignment is weighted combination
2118:         alignment = (ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1915_21", 0.6) * cluster_c
oherence) + (ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1915_49", 0.4) * validation_rate)
2119:
2120:         logger.debug(
2121:             f"Strategic alignment: cluster_coherence={cluster_coherence:.4f}, "
2122:             f"validation_rate={validation_rate:.4f}, alignment={alignment:.4f}"
2123:         )
2124:
2125:         return alignment
2126:
2127:     def apply_rubric_thresholds(
2128:         self,
2129:         score: float,
2130:         thresholds: dict[str, float] | None = None
2131:     ) -> str:
2132:         """
2133:         Apply macro-level rubric thresholds.
2134:
2135:         Args:
2136:             score: Aggregated macro score (0-3 range)
2137:             thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO, ACEPTABLE)
2138:                         Each value should be a normalized threshold (0-1 range)
2139:
2140:         Returns:
2141:             Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
2142:         """
2143:         # Clamp score to valid range [0, 3]
2144:         clamped_score = max(ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1941_28", 0.0), min
(3.0, score))
2145:
2146:         # Normalize to 0-1 range
2147:         normalized_score = clamped_score / 3.0
2148:
2149:         # Use provided thresholds or defaults
2150:         if thresholds:
2151:             excellent_threshold = thresholds.get('EXCELENTE', ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence"
```

```
, "auto_param_L1948_62", 0.85))
 2152:             good_threshold = thresholds.get('BUENO', ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_p
aram_L1949_53", 0.70))
 2153:             acceptable_threshold = thresholds.get('ACEPTABLE', ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence
", "auto_param_L1950_63", 0.55))
 2154:         else:
 2155:             excellent_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "excellent_threshold", 0.8
5) # Refactored
 2156:             good_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "good_threshold", 0.7) # Refact
ored
 2157:             acceptable_threshold = ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "acceptable_threshold", 0
.55) # Refactored
 2158:
 2159:         # Apply thresholds
 2160:         if normalized_score >= excellent_threshold:
 2161:             quality = "EXCELENTE"
 2162:         elif normalized_score >= good_threshold:
 2163:             quality = "BUENO"
 2164:         elif normalized_score >= acceptable_threshold:
 2165:             quality = "ACEPTABLE"
 2166:         else:
 2167:             quality = "INSUFICIENTE"
 2168:
 2169:         logger.debug(
 2170:             f"Macro rubric applied: score={score:.4f}, "
 2171:             f"normalized={normalized_score:.4f}, quality={quality}"
 2172:         )
 2173:
 2174:         return quality
 2175:
 2176:     def evaluate_macro(
 2177:         self,
 2178:         cluster_scores: list[ClusterScore],
 2179:         area_scores: list[AreaScore],
 2180:         dimension_scores: list[DimensionScore]
 2181:     ) -> MacroScore:
 2182:         """
 2183:         Perform holistic macro evaluation (Q305).
 2184:
 2185:         Args:
 2186:             cluster_scores: List of cluster scores (MESO level)
 2187:             area_scores: List of area scores
 2188:             dimension_scores: List of dimension scores
 2189:
 2190:         Returns:
 2191:             MacroScore with holistic evaluation
 2192:         """
 2193:         logger.info("Performing macro holistic evaluation (Q305)")
 2194:
 2195:         validation_details = {}
 2196:
 2197:         if not cluster_scores:
 2198:             logger.error("No cluster scores available for macro evaluation")
 2199:             return MacroScore(
 2200:                 score=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1997_22", 0.0),
 2201:                 quality_level="INSUFICIENTE",
```

```
                cross_cutting_coherence=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L1999_40
", 0.0),
                systemic_gaps=[],
                strategic_alignment=ParameterLoaderV2.get("farfan_core.processing.aggregation.ClusterAggregator.analyze_coherence", "auto_param_L2001_36", 0
.0),
                cluster_scores=[],
                validation_passed=False,
                validation_details={"error": "No clusters", "type": "empty"}
            )

        # Calculate cross-cutting coherence
        cross_cutting_coherence = self.calculate_cross_cutting_coherence(cluster_scores)
        validation_details["coherence"] = {
            "value": cross_cutting_coherence,
            "clusters": len(cluster_scores)
        }

        # Identify systemic gaps
        systemic_gaps = self.identify_systemic_gaps(area_scores)
        validation_details["gaps"] = {
            "count": len(systemic_gaps),
            "areas": systemic_gaps
        }

        # Assess strategic alignment
        strategic_alignment = self.assess_strategic_alignment(
            cluster_scores,
            dimension_scores
        )
        validation_details["alignment"] = {
            "value": strategic_alignment
        }

        # Calculate overall macro score (weighted average of clusters)
        macro_score = self._calculate_macro_score(cluster_scores)

        # Apply quality rubric
        quality_level = self.apply_rubric_thresholds(macro_score)
        validation_details["rubric"] = {
            "score": macro_score,
            "quality_level": quality_level
        }

        logger.info(
            f"â\234\223 Macro evaluation (Q305): score={macro_score:.4f}, "
            f"quality={quality_level}, coherence={cross_cutting_coherence:.4f}, "
            f"alignment={strategic_alignment:.4f}, gaps={len(systemic_gaps)}"
        )

        return MacroScore(
            score=macro_score,
            quality_level=quality_level,
            cross_cutting_coherence=cross_cutting_coherence,
            systemic_gaps=systemic_gaps,
            strategic_alignment=strategic_alignment,
            cluster_scores=cluster_scores,
```

```
            validation_passed=True,
            validation_details=validation_details
        )

    @calibrated_method("farfan_core.processing.aggregation.MacroAggregator._calculate_macro_score")
    def _calculate_macro_score(self, cluster_scores: list[ClusterScore]) -> float:
        weights = self.aggregation_settings.macro_cluster_weights
        if not cluster_scores:
            return ParameterLoaderV2.get("farfan_core.processing.aggregation.MacroAggregator._calculate_macro_score", "auto_param_L2061_19", 0.0)
        if not weights:
            return sum(c.score for c in cluster_scores) / len(cluster_scores)

        resolved_weights: list[float] = []
        for cluster in cluster_scores:
            weight = weights.get(cluster.cluster_id)
            if weight is None:
                logger.debug(
                    "Missing macro weight for cluster %s â\200\223 falling back to equal weights",
                    cluster.cluster_id,
                )
                return sum(c.score for c in cluster_scores) / len(cluster_scores)
            resolved_weights.append(weight)

        total = sum(resolved_weights)
        if total <= 0:
            return sum(c.score for c in cluster_scores) / len(cluster_scores)

        normalized = [w / total for w in resolved_weights]
        return sum(
            cluster.score * weight
            for cluster, weight in zip(cluster_scores, normalized, strict=False)
        )



================================================================================
FILE: src/farfan_pipeline/processing/aggregation_provenance.py
================================================================================

"""
Aggregation Provenance System - DAG-based Lineage Tracking

This module provides full provenance tracking for the hierarchical aggregation pipeline.
It implements W3C PROV-compliant directed acyclic graphs (DAGs) to capture:
- Data lineage: Which micro-questions contributed to which macro-scores
- Operation tracking: How aggregation operations transformed data
- Sensitivity analysis: Which inputs have highest impact on outputs
- Counterfactual reasoning: What-if analysis for policy decisions

Architecture:
- ProvenanceNode: Immutable record of a score at any aggregation level
- AggregationDAG: NetworkX-based graph with attribution methods
- SHAPAttribution: Shapley value computation for feature importance

References:
- W3C PROV: https://www.w3.org/TR/prov-overview/
```

```
- Shapley values: Shapley, L.S. (1953). "A value for n-person games"
- NetworkX: Hagberg et al. (2008). "Exploring network structure, dynamics, and function"
"""

from __future__ import annotations

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

import networkx as nx
import numpy as np

logger = logging.getLogger(__name__)


@dataclass(frozen=True)
class ProvenanceNode:
    """
    Immutable provenance record for a score node in the aggregation DAG.

    Attributes:
        node_id: Unique identifier (e.g., "Q001", "DIM01_PA05", "CLUSTER_MESO_1")
        level: Abstraction level in hierarchy
        score: Numeric score value
        quality_level: Quality classification (EXCELENTE, BUENO, etc.)
        timestamp: ISO timestamp of computation
        metadata: Additional context (weights, confidence, etc.)
    """
    node_id: str
    level: str  # "micro", "dimension", "area", "cluster", "macro"
    score: float
    quality_level: str
    timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
    metadata: dict[str, Any] = field(default_factory=dict)

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

    def compute_hash(self) -> str:
        """Compute deterministic hash for reproducibility."""
        content = json.dumps(
            {
                "node_id": self.node_id,
                "level": self.level,
                "score": self.score,
                "quality_level": self.quality_level,
            },
            sort_keys=True,
        )
        return hashlib.sha256(content.encode("utf-8")).hexdigest()[:16]
```

```
2368:
2369:
2370: @dataclass
2371: class AggregationEdge:
2372:     """
2373:     Edge in the provenance DAG representing an aggregation operation.
2374:
2375:     Attributes:
2376:         source_id: Parent node contributing to aggregation
2377:         target_id: Child node receiving aggregated score
2378:         operation: Type of aggregation (weighted_average, choquet, max, etc.)
2379:         weight: Contribution weight (for weighted operations)
2380:         timestamp: When this edge was created
2381:     """
2382:     source_id: str
2383:     target_id: str
2384:     operation: str
2385:     weight: float
2386:     timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat())
2387:     metadata: dict[str, Any] = field(default_factory=dict)
2388:
2389:
2390: class AggregationDAG:
2391:     """
2392:     Directed Acyclic Graph for full provenance tracking of aggregation pipeline.
2393:
2394:     This class maintains the complete lineage of how micro-question scores
2395:     propagate through dimension â\206\222 area â\206\222 cluster â\206\222 macro aggregation.
2396:
2397:     Features:
2398:     - Cycle detection (enforces DAG property)
2399:     - Topological sorting for dependency resolution
2400:     - Shapley value attribution for sensitivity analysis
2401:     - GraphML export for visualization in Gephi/Cytoscape
2402:     - W3C PROV-compliant serialization
2403:     """
2404:
2405:     def __init__(self):
2406:         """Initialize empty DAG."""
2407:         self.graph: nx.DiGraph = nx.DiGraph()
2408:         self.nodes: dict[str, ProvenanceNode] = {}
2409:         self.edges: list[AggregationEdge] = []
2410:         logger.info("AggregationDAG initialized")
2411:
2412:     def add_node(self, node: ProvenanceNode) -> None:
2413:         """
2414:         Add a provenance node to the DAG.
2415:
2416:         Args:
2417:             node: ProvenanceNode to add
2418:
2419:         Raises:
2420:             ValueError: If node_id already exists
2421:         """
2422:         if node.node_id in self.nodes:
2423:             logger.warning(f"Node {node.node_id} already exists, skipping")
```

```
            return

        self.nodes[node.node_id] = node
        self.graph.add_node(
            node.node_id,
            level=node.level,
            score=node.score,
            quality=node.quality_level,
            timestamp=node.timestamp,
            metadata=node.metadata,
        )
        logger.debug(f"Added node {node.node_id} (level={node.level}, score={node.score:.2f})")

    def add_aggregation_edge(
        self,
        source_ids: list[str],
        target_id: str,
        operation: str,
        weights: list[float],
        metadata: dict[str, Any] | None = None,
    ) -> None:
        """
        Record an aggregation operation: sources â\206\222 target.

        Args:
            source_ids: List of source node IDs (e.g., micro-questions)
            target_id: Target node ID (e.g., dimension score)
            operation: Aggregation type (weighted_average, choquet, etc.)
            weights: Contribution weights for each source
            metadata: Additional operation metadata

        Raises:
            ValueError: If weights don't match sources or cycle detected
        """
        if len(source_ids) != len(weights):
            raise ValueError(
                f"Mismatch: {len(source_ids)} sources but {len(weights)} weights"
            )

        if metadata is None:
            metadata = {}

        for source_id, weight in zip(source_ids, weights):
            if source_id not in self.graph:
                logger.warning(f"Source node {source_id} not found, adding placeholder")
                self.graph.add_node(source_id, level="unknown", score=0.0)

            if target_id not in self.graph:
                logger.warning(f"Target node {target_id} not found, adding placeholder")
                self.graph.add_node(target_id, level="unknown", score=0.0)

            edge = AggregationEdge(
                source_id=source_id,
                target_id=target_id,
                operation=operation,
                weight=weight,
```

```
                metadata=metadata,
            )
            self.edges.append(edge)

            self.graph.add_edge(
                source_id,
                target_id,
                operation=operation,
                weight=weight,
                timestamp=edge.timestamp,
                metadata=metadata,
            )

        # Verify DAG property (no cycles)
        if not nx.is_directed_acyclic_graph(self.graph):
            # Rollback last edges
            for source_id in source_ids:
                if self.graph.has_edge(source_id, target_id):
                    self.graph.remove_edge(source_id, target_id)
            raise ValueError(f"Adding edges {source_ids} â\206\222 {target_id} would create a cycle")

        logger.info(
            f"Added aggregation: {len(source_ids)} sources â\206\222 {target_id} "
            f"(operation={operation})"
        )

    def trace_lineage(self, target_id: str) -> dict[str, Any]:
        """
        Trace complete lineage of a target node.

        Returns all ancestor nodes, the aggregation path, and sensitivity metrics.

        Args:
            target_id: Node ID to trace

        Returns:
            Dictionary with:
            - ancestors: Set of all ancestor node IDs
            - path: Topologically sorted path from sources to target
            - depth: Maximum path length from any micro-question
            - breadth: Number of unique micro-questions contributing

        Raises:
            ValueError: If target_id not in graph
        """
        if target_id not in self.graph:
            raise ValueError(f"Node {target_id} not found in DAG")

        ancestors = nx.ancestors(self.graph, target_id)
        subgraph = self.graph.subgraph(ancestors | {target_id})

        # Compute metrics
        depth = nx.dag_longest_path_length(subgraph) if subgraph.nodes else 0

        # Count micro-questions (level="micro")
        micro_nodes = [
```

```
            n for n in ancestors
            if self.graph.nodes[n].get("level") == "micro"
        ]

        # Get topological path
        topo_path = list(nx.topological_sort(subgraph))

        return {
            "target_id": target_id,
            "ancestor_count": len(ancestors),
            "ancestors": sorted(ancestors),
            "topological_path": topo_path,
            "depth": depth,
            "micro_question_count": len(micro_nodes),
            "micro_questions": sorted(micro_nodes),
        }

    def compute_shapley_attribution(self, target_id: str) -> dict[str, float]:
        """
        Compute Shapley values for feature attribution.

        Shapley values represent the marginal contribution of each source node
        to the target score, accounting for all possible coalitions.

        Args:
            target_id: Node to attribute

        Returns:
            Dictionary mapping source node IDs to Shapley values (sum = target score)

        Note:
            This is an exact computation for weighted averages. For non-linear
            aggregations (Choquet), this uses kernel SHAP approximation.
        """
        if target_id not in self.graph:
            raise ValueError(f"Node {target_id} not found in DAG")

        # Get direct predecessors (sources)
        predecessors = list(self.graph.predecessors(target_id))

        if not predecessors:
            logger.warning(f"Node {target_id} has no predecessors")
            return {}

        # Extract weights and scores
        weights = []
        scores = []
        for pred in predecessors:
            edge_data = self.graph.get_edge_data(pred, target_id)
            weights.append(edge_data.get("weight", 0.0))
            scores.append(self.graph.nodes[pred].get("score", 0.0))

        weights = np.array(weights)
        scores = np.array(scores)

        # For weighted average, Shapley value = weight Ã\227 score
```

```
        # This is exact because weighted average is a linear function
        shapley_values = weights * scores

        # Normalize to sum to target score
        target_score = self.graph.nodes[target_id].get("score", 0.0)
        if np.sum(shapley_values) > 0:
            shapley_values = shapley_values * (target_score / np.sum(shapley_values))

        attribution = {
            pred: float(shap_val)
            for pred, shap_val in zip(predecessors, shapley_values)
        }

        logger.debug(
            f"Shapley attribution for {target_id}: "
            f"{len(attribution)} sources, sum={sum(attribution.values()):.4f}"
        )

        return attribution

    def get_critical_path(self, target_id: str, top_k: int = 5) -> list[tuple[str, float]]:
        """
        Identify the most critical source nodes for a target.

        Uses Shapley values to rank sources by importance.

        Args:
            target_id: Target node
            top_k: Number of top sources to return

        Returns:
            List of (node_id, shapley_value) tuples, sorted by importance
        """
        attribution = self.compute_shapley_attribution(target_id)

        # Sort by absolute Shapley value (handle negative contributions)
        sorted_attribution = sorted(
            attribution.items(),
            key=lambda x: abs(x[1]),
            reverse=True,
        )

        return sorted_attribution[:top_k]

    def export_graphml(self, path: str) -> None:
        """
        Export DAG to GraphML format for visualization.

        Compatible with:
        - Gephi: https://gephi.org/
        - Cytoscape: https://cytoscape.org/
        - yEd: https://www.yworks.com/products/yed

        Args:
            path: Output file path (e.g., "aggregation_dag.graphml")
        """
```

```
        # Create a copy with serialized dict attributes (GraphML doesn't support dicts)
        g_copy = self.graph.copy()
        for node, data in g_copy.nodes(data=True):
            if "metadata" in data and isinstance(data["metadata"], dict):
                data["metadata_json"] = json.dumps(data["metadata"])
                del data["metadata"]
        for u, v, data in g_copy.edges(data=True):
            if "metadata" in data and isinstance(data["metadata"], dict):
                data["metadata_json"] = json.dumps(data["metadata"])
                del data["metadata"]
            if "weights" in data and isinstance(data["weights"], list):
                data["weights_json"] = json.dumps(data["weights"])
                del data["weights"]
        nx.write_graphml(g_copy, path)
        logger.info(
            f"Exported DAG to {path}: "
            f"{self.graph.number_of_nodes()} nodes, "
            f"{self.graph.number_of_edges()} edges"
        )

    def export_prov_json(self, path: str) -> None:
        """
        Export to W3C PROV-JSON format.

        Spec: https://www.w3.org/Submission/prov-json/

        Args:
            path: Output file path (e.g., "provenance.json")
        """
        prov_doc = {
            "prefix": {
                "prov": "http://www.w3.org/ns/prov#",
                "farfan": "http://farfan.org/ns/aggregation#",
            },
            "entity": {},
            "activity": {},
            "wasGeneratedBy": {},
            "used": {},
        }

        # Entities: All nodes
        for node_id, node_data in self.graph.nodes(data=True):
            prov_doc["entity"][node_id] = {
                "prov:type": "farfan:ScoreEntity",
                "farfan:level": node_data.get("level"),
                "farfan:score": node_data.get("score"),
                "farfan:quality": node_data.get("quality"),
                "prov:generatedAtTime": node_data.get("timestamp"),
            }

        # Activities: Aggregation operations
        for idx, edge in enumerate(self.edges):
            activity_id = f"agg_{idx}"
            prov_doc["activity"][activity_id] = {
                "prov:type": f"farfan:{edge.operation}",
                "farfan:weight": edge.weight,
```

```
                "prov:startedAtTime": edge.timestamp,
            }

            # wasGeneratedBy: target was generated by this activity
            if edge.target_id not in prov_doc["wasGeneratedBy"]:
                prov_doc["wasGeneratedBy"][edge.target_id] = []
            prov_doc["wasGeneratedBy"][edge.target_id].append(activity_id)

            # used: activity used source
            if activity_id not in prov_doc["used"]:
                prov_doc["used"][activity_id] = []
            prov_doc["used"][activity_id].append(edge.source_id)

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

        logger.info(f"Exported PROV-JSON to {path}")

    def get_statistics(self) -> dict[str, Any]:
        """
        Get DAG statistics for monitoring and validation.

        Returns:
            Dictionary with graph metrics
        """
        return {
            "node_count": self.graph.number_of_nodes(),
            "edge_count": self.graph.number_of_edges(),
            "max_depth": nx.dag_longest_path_length(self.graph) if self.graph.nodes else 0,
            "is_dag": nx.is_directed_acyclic_graph(self.graph),
            "weakly_connected_components": nx.number_weakly_connected_components(self.graph),
            "nodes_by_level": self._count_by_level(),
        }

    def _count_by_level(self) -> dict[str, int]:
        """Count nodes by abstraction level."""
        counts = defaultdict(int)
        for node_id in self.graph.nodes:
            level = self.graph.nodes[node_id].get("level", "unknown")
            counts[level] += 1
        return dict(counts)


__all__ = [
    "ProvenanceNode",
    "AggregationEdge",
    "AggregationDAG",
]



================================================================================
FILE: src/farfan_pipeline/processing/choquet_adapter.py
================================================================================

"""
```

```
Choquet Integral Adapter for Processing Aggregation

This module adapts the calibration ChoquetAggregator for use in processing-level
score aggregation. It handles the impedance mismatch between:
- Calibration layer scores (8 layers: @b, @chain, @q, @d, @p, @C, @u, @m)
- Processing dimension/area scores (numeric scores with weights)

Key adaptations:
- Maps dimension/question indices to pseudo-layer IDs
- Converts score lists + weights to layer score dictionaries
- Preserves non-linear interaction semantics from Choquet integral

References:
- Grabisch, M. (2016). "Set Functions, Games and Capacities in Decision Making"
- Choquet, G. (1953). "Theory of capacities"
"""

from __future__ import annotations

import logging
from typing import Any

import numpy as np

logger = logging.getLogger(__name__)


class ChoquetProcessingAdapter:
    """
    Adapter for using Choquet integral in processing aggregation.

    This class bridges the gap between calibration (layer-based) and
    processing (score list) contexts.
    """

    def __init__(
        self,
        linear_weights: dict[str, float] | None = None,
        interaction_weights: dict[tuple[str, str], float] | None = None,
    ):
        """
        Initialize Choquet adapter.

        Args:
            linear_weights: Weights for linear terms (default: uniform)
            interaction_weights: Weights for interaction terms (default: none)
        """
        self.linear_weights = linear_weights or {}
        self.interaction_weights = interaction_weights or {}
        logger.info(
            f"ChoquetProcessingAdapter initialized: "
            f"{len(self.linear_weights)} linear, "
            f"{len(self.interaction_weights)} interaction terms"
        )

    def aggregate(
```

```
2816:         self,
2817:         scores: list[float],
2818:         weights: list[float] | None = None,
2819:         interaction_pairs: list[tuple[int, int]] | None = None,
2820:     ) -> float:
2821:         """
2822:         Aggregate scores using Choquet 2-additive integral.
2823:
2824:         Formula:
2825:             C(x) = Î£ w_i * x_i + Î£ w_ij * min(x_i, x_j)
2826:
2827:         Where:
2828:         - First sum: Linear terms (weighted individual scores)
2829:         - Second sum: Interaction terms (synergy/redundancy between pairs)
2830:
2831:         Args:
2832:             scores: List of numeric scores
2833:             weights: Linear weights for each score (default: uniform)
2834:             interaction_pairs: List of (i, j) index pairs for interactions
2835:
2836:         Returns:
2837:             Aggregated score
2838:
2839:         Raises:
2840:             ValueError: If weights mismatch or scores empty
2841:         """
2842:         if not scores:
2843:             raise ValueError("Cannot aggregate empty score list")
2844:
2845:         n = len(scores)
2846:         scores_arr = np.array(scores, dtype=np.float64)
2847:
2848:         # Default: uniform weights
2849:         if weights is None:
2850:             weights_arr = np.ones(n) / n
2851:         else:
2852:             if len(weights) != n:
2853:                 raise ValueError(f"Weight count {len(weights)} != score count {n}")
2854:             weights_arr = np.array(weights, dtype=np.float64)
2855:             # Normalize to sum to 1.0
2856:             weights_arr = weights_arr / np.sum(weights_arr)
2857:
2858:         # STEP 1: Linear contribution
2859:         # C_linear = Î£ w_i * x_i
2860:         linear_contrib = float(np.sum(weights_arr * scores_arr))
2861:
2862:         logger.debug(f"Choquet linear contribution: {linear_contrib:.4f}")
2863:
2864:         # STEP 2: Interaction contribution
2865:         # C_interaction = Î£ w_ij * min(x_i, x_j)
2866:         interaction_contrib = 0.0
2867:
2868:         if interaction_pairs:
2869:             for i, j in interaction_pairs:
2870:                 if i >= n or j >= n:
2871:                     logger.warning(f"Interaction pair ({i}, {j}) out of bounds (n={n})")
```

```
                    continue

                # Interaction weight (default: small positive for synergy)
                pair_key = (str(i), str(j))
                w_ij = self.interaction_weights.get(pair_key, 0.05)

                # Choquet uses min() for weakest-link principle
                min_score = min(scores_arr[i], scores_arr[j])
                contribution = w_ij * min_score
                interaction_contrib += contribution

                logger.debug(
                    f"Interaction ({i}, {j}): "
                    f"min({scores_arr[i]:.2f}, {scores_arr[j]:.2f}) = {min_score:.2f}, "
                    f"w={w_ij:.3f}, contrib={contribution:.4f}"
                )

        logger.debug(f"Choquet interaction contribution: {interaction_contrib:.4f}")

        # STEP 3: Total score
        total = linear_contrib + interaction_contrib

        logger.info(
            f"Choquet aggregation: {n} inputs â\206\222 {total:.4f} "
            f"(linear={linear_contrib:.4f}, interaction={interaction_contrib:.4f})"
        )

        return float(total)

    @staticmethod
    def detect_interactions(
        scores: list[float],
        weights: list[float],
        correlation_threshold: float = 0.7,
    ) -> list[tuple[int, int]]:
        """
        Auto-detect interaction pairs based on score correlation.

        Pairs with high correlation should be penalized (redundancy),
        pairs with negative correlation should be rewarded (complementarity).

        Args:
            scores: Score values
            weights: Score weights
            correlation_threshold: Min correlation for interaction

        Returns:
            List of (i, j) pairs with significant interactions

        Note:
            This is a heuristic. For production, use domain knowledge to
            specify interactions explicitly.
        """
        n = len(scores)
        pairs = []

```

```
        # For now, use a simple heuristic: top-weighted pairs
        # In practice, you'd analyze historical data for correlations
        sorted_indices = np.argsort(weights)[::-1]  # Descending

        # Consider top 3 interactions among high-weight inputs
        if n >= 2:
            for i in range(min(3, n - 1)):
                idx_i = sorted_indices[i]
                idx_j = sorted_indices[i + 1]
                pairs.append((int(idx_i), int(idx_j)))

        logger.debug(f"Auto-detected {len(pairs)} interaction pairs")
        return pairs


def create_default_choquet_adapter(n_inputs: int) -> ChoquetProcessingAdapter:
    """
    Create a Choquet adapter with sensible defaults.

    Default configuration:
    - Uniform linear weights (each input gets 1/n)
    - Small interaction weights (0.05) for adjacent pairs
    - Synergistic interactions (positive weights)

    Args:
        n_inputs: Number of input scores

    Returns:
        Configured ChoquetProcessingAdapter
    """
    # Linear weights: uniform
    linear_weights = {str(i): 1.0 / n_inputs for i in range(n_inputs)}

    # Interaction weights: adjacent pairs only (locality assumption)
    interaction_weights = {}
    for i in range(n_inputs - 1):
        pair_key = (str(i), str(i + 1))
        interaction_weights[pair_key] = 0.05  # Small synergy

    logger.info(
        f"Created default Choquet adapter: "
        f"{n_inputs} inputs, {len(interaction_weights)} interactions"
    )

    return ChoquetProcessingAdapter(
        linear_weights=linear_weights,
        interaction_weights=interaction_weights,
    )


__all__ = [
    "ChoquetProcessingAdapter",
    "create_default_choquet_adapter",
]


```

```
2984:
2985: ================================================================================
2986: FILE: src/farfan_pipeline/processing/converter.py
2987: ================================================================================
2988:
2989: """
2990: SmartChunk to CanonPolicyPackage Converter
2991: ==========================================
2992:
2993: This module provides the critical bridge layer between the SPC (Smart Policy Chunks)
2994: phase-one output and the CanonPolicyPackage format expected by the orchestrator.
2995:
2996: Architecture:
2997:     SmartPolicyChunk (from StrategicChunkingSystem)
2998:         â\206\223
2999:     SmartChunkConverter (this module)
3000:         â\206\223
3001:     CanonPolicyPackage (for SPCAdapter and Orchestrator)
3002:
3003: Key Responsibilities:
3004: 1. Convert SmartPolicyChunk dataclass to Chunk dataclass
3005: 2. Map chunk_type (8 types) to resolution (MICRO/MESO/MACRO)
3006: 3. Extract policy/time/geo facets from SPC rich data
3007: 4. Build ChunkGraph with edges from related_chunks
3008: 5. Preserve SPC rich data in metadata for executor access
3009: 6. Generate quality metrics and integrity index
3010: """
3011:
3012: from __future__ import annotations
3013:
3014: import hashlib
3015: import json
3016: import logging
3017: from typing import TYPE_CHECKING, Any
3018:
3019: from farfan_pipeline.core.calibration.decorators import calibrated_method
3020: from farfan_pipeline.processing.cpp_ingestion.models import (
3021:     KPI,
3022:     Budget,
3023:     CanonPolicyPackage,
3024:     Chunk,
3025:     ChunkGraph,
3026:     ChunkResolution,
3027:     Confidence,
3028:     Entity,
3029:     GeoFacet,
3030:     IntegrityIndex,
3031:     PolicyFacet,
3032:     PolicyManifest,
3033:     ProvenanceMap,
3034:     QualityMetrics,
3035:     TextSpan,
3036:     TimeFacet,
3037: )
3038:
3039: if TYPE_CHECKING:
```

```
    # Avoid runtime import of SmartPolicyChunk (heavy dependencies)
    from typing import Protocol

    class SmartPolicyChunkProtocol(Protocol):
        """Protocol for SmartPolicyChunk to avoid circular imports"""
        chunk_id: str
        document_id: str
        content_hash: str
        text: str
        normalized_text: str
        semantic_density: float
        section_hierarchy: list[str]
        document_position: tuple[int, int]
        chunk_type: Any  # ChunkType enum
        causal_chain: list[Any]
        policy_entities: list[Any]
        related_chunks: list[tuple[str, float]]
        confidence_metrics: dict[str, float]
        coherence_score: float
        completeness_index: float
        strategic_importance: float

logger = logging.getLogger(__name__)


class SmartChunkConverter:
    """
    Converts SmartPolicyChunk instances to CanonPolicyPackage format.

    This converter is the critical bridge that enables SPC phase-one output
    to be consumed by the orchestrator and its executors.
    """

    # Mapping from ChunkType to ChunkResolution
    CHUNK_TYPE_TO_RESOLUTION = {
        'DIAGNOSTICO': ChunkResolution.MESO,
        'ESTRATEGIA': ChunkResolution.MACRO,
        'METRICA': ChunkResolution.MICRO,
        'FINANCIERO': ChunkResolution.MICRO,
        'NORMATIVO': ChunkResolution.MESO,
        'OPERATIVO': ChunkResolution.MICRO,
        'EVALUACION': ChunkResolution.MESO,
        'MIXTO': ChunkResolution.MESO,
    }

    def __init__(self) -> None:
        """Initialize the converter."""
        self.logger = logging.getLogger(self.__class__.__name__)

    def convert_to_canon_package(
        self,
        smart_chunks: list[Any],  # List[SmartPolicyChunk]
        document_metadata: dict[str, Any]
    ) -> CanonPolicyPackage:
        """
        Convert list of SmartPolicyChunk to CanonPolicyPackage.
```

```
3096:
3097:         Args:
3098:             smart_chunks: List of SmartPolicyChunk instances from StrategicChunkingSystem
3099:             document_metadata: Document-level metadata (id, title, version, etc.)
3100:
3101:         Returns:
3102:             CanonPolicyPackage ready for orchestrator consumption
3103:
3104:         Raises:
3105:             ValueError: If smart_chunks is empty or invalid
3106:         """
3107:         # Defensive validation: ensure smart_chunks is non-empty
3108:         if not smart_chunks or len(smart_chunks) == 0:
3109:             raise ValueError(
3110:                 "Cannot convert empty smart_chunks list to CanonPolicyPackage. "
3111:                 "Minimum 1 chunk required from StrategicChunkingSystem."
3112:             )
3113:
3114:         # Defensive validation: check critical attributes on first chunk
3115:         first_chunk = smart_chunks[0]
3116:         required_attrs = ['chunk_id', 'document_id', 'text', 'document_position', 'chunk_type']
3117:         missing_attrs = [attr for attr in required_attrs if not hasattr(first_chunk, attr)]
3118:
3119:         if missing_attrs:
3120:             raise ValueError(
3121:                 f"SmartPolicyChunk missing critical attributes: {missing_attrs}. "
3122:                 f"Ensure StrategicChunkingSystem produced valid SmartPolicyChunk instances. "
3123:                 f"Chunk type: {type(first_chunk)}"
3124:             )
3125:
3126:         self.logger.info(f"Converting {len(smart_chunks)} SmartPolicyChunks to CanonPolicyPackage")
3127:
3128:         # Build ChunkGraph
3129:         chunk_graph = ChunkGraph()
3130:
3131:         # Convert each SmartPolicyChunk to Chunk
3132:         chunk_hashes = {}
3133:         all_axes = set()
3134:         all_programs = set()
3135:         all_projects = set()
3136:         all_years = set()
3137:         all_territories = set()
3138:
3139:         for smart_chunk in smart_chunks:
3140:             # Convert to Chunk
3141:             chunk = self._convert_smart_chunk_to_chunk(smart_chunk)
3142:
3143:             # Add to ChunkGraph
3144:             chunk_graph.chunks[chunk.id] = chunk
3145:             chunk_hashes[chunk.id] = chunk.bytes_hash
3146:
3147:             # Collect manifest data
3148:             all_axes.update(chunk.policy_facets.axes)
3149:             all_programs.update(chunk.policy_facets.programs)
3150:             all_projects.update(chunk.policy_facets.projects)
3151:             all_years.update(chunk.time_facets.years)
```

```
            all_territories.update(chunk.geo_facets.territories)

        # Build edges from related_chunks
        for smart_chunk in smart_chunks:
            if hasattr(smart_chunk, 'related_chunks') and smart_chunk.related_chunks:
                for related_id, similarity in smart_chunk.related_chunks[:5]:  # Top 5
                    # Only add edge if target chunk exists
                    if related_id in chunk_graph.chunks:
                        edge = (smart_chunk.chunk_id, related_id, f"semantic_similarity_{similarity:.2f}")
                        chunk_graph.edges.append(edge)

        self.logger.info(f"Built ChunkGraph with {len(chunk_graph.chunks)} chunks and {len(chunk_graph.edges)} edges")

        # Create PolicyManifest
        policy_manifest = PolicyManifest(
            axes=sorted(all_axes),
            programs=sorted(all_programs),
            projects=sorted(all_projects),
            years=sorted(all_years),
            territories=sorted(all_territories),
            indicators=[],  # Would extract from KPIs if available
            budget_rows=sum(1 for c in chunk_graph.chunks.values() if c.budget is not None)
        )

        # Calculate QualityMetrics
        quality_metrics = self._calculate_quality_metrics(smart_chunks, chunk_graph)

        # Generate IntegrityIndex
        integrity_index = self._generate_integrity_index(chunk_hashes, document_metadata)

        # Preserve SPC rich data in metadata
        enriched_metadata = self._preserve_spc_rich_data(smart_chunks, document_metadata)

        # Build CanonPolicyPackage
        canon_package = CanonPolicyPackage(
            schema_version="SPC-2025.1",
            chunk_graph=chunk_graph,
            policy_manifest=policy_manifest,
            quality_metrics=quality_metrics,
            integrity_index=integrity_index,
            metadata=enriched_metadata
        )

        self.logger.info("Successfully converted to CanonPolicyPackage")
        return canon_package

    @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._convert_smart_chunk_to_chunk")
    def _convert_smart_chunk_to_chunk(self, smart_chunk: Any) -> Chunk:
        """
        Convert a single SmartPolicyChunk to Chunk.

        Maps fields from SPC rich format to orchestrator-compatible format.
        """
        # Determine resolution from chunk_type
        chunk_type_str = smart_chunk.chunk_type.value if hasattr(smart_chunk.chunk_type, 'value') else str(smart_chunk.chunk_type)
        resolution = self.CHUNK_TYPE_TO_RESOLUTION.get(chunk_type_str.upper(), ChunkResolution.MESO)
```

```
3208:
3209:         # Extract policy facets
3210:         policy_facets = self._extract_policy_facets(smart_chunk)
3211:
3212:         # Extract time facets
3213:         time_facets = self._extract_time_facets(smart_chunk)
3214:
3215:         # Extract geo facets
3216:         geo_facets = self._extract_geo_facets(smart_chunk)
3217:
3218:         # Build confidence from SPC metrics
3219:         confidence = Confidence(
3220:             layout=ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._convert_smart_chunk_to_chunk", "auto_param_L232_19", 1.0),  # SPC doesn't distinguish these
3221:             ocr=smart_chunk.confidence_metrics.get('extraction_confidence', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._convert_smart_chunk_to_chunk", "auto_param_L233_76", 0.95)),
3222:             typing=smart_chunk.coherence_score
3223:         )
3224:
3225:         # Create provenance
3226:         provenance = self._build_provenance(smart_chunk)
3227:
3228:         # Extract entities
3229:         entities = self._extract_entities(smart_chunk)
3230:
3231:         # Extract budget if available
3232:         budget = self._extract_budget(smart_chunk)
3233:
3234:         # Extract KPI if available
3235:         kpi = self._extract_kpi(smart_chunk)
3236:
3237:         # Build Chunk
3238:         return Chunk(
3239:             id=smart_chunk.chunk_id,
3240:             text=smart_chunk.text,
3241:             text_span=TextSpan(
3242:                 start=smart_chunk.document_position[0],
3243:                 end=smart_chunk.document_position[1]
3244:             ),
3245:             resolution=resolution,
3246:             bytes_hash=smart_chunk.content_hash,
3247:             policy_area_id=getattr(smart_chunk, 'policy_area_id', None),  # PA01-PA10
3248:             dimension_id=getattr(smart_chunk, 'dimension_id', None),      # DIM01-DIM06
3249:             policy_facets=policy_facets,
3250:             time_facets=time_facets,
3251:             geo_facets=geo_facets,
3252:             confidence=confidence,
3253:             provenance=provenance,
3254:             budget=budget,
3255:             kpi=kpi,
3256:             entities=entities
3257:         )
3258:
3259:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_policy_facets")
3260:     def _extract_policy_facets(self, smart_chunk: Any) -> PolicyFacet:
3261:         """Extract policy facets from SPC strategic_context and section_hierarchy."""
```

```
        axes = []
        programs = []
        projects = []

        # Extract from strategic_context if available
        if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
            ctx = smart_chunk.strategic_context
            # strategic_context might have policy_intent, implementation_phase
            if hasattr(ctx, 'policy_intent'):
                axes.append(ctx.policy_intent[:50])  # Truncate if too long
            if hasattr(ctx, 'implementation_phase'):
                programs.append(ctx.implementation_phase[:50])

        # Extract from section_hierarchy
        if hasattr(smart_chunk, 'section_hierarchy') and smart_chunk.section_hierarchy:
            hierarchy = smart_chunk.section_hierarchy
            if len(hierarchy) > 0:
                axes.append(hierarchy[0])  # Top-level = axis
            if len(hierarchy) > 1:
                programs.append(hierarchy[1])  # Second level = program
            if len(hierarchy) > 2:
                projects.append(hierarchy[2])  # Third level = project

        return PolicyFacet(
            axes=axes[:3],  # Limit to avoid bloat
            programs=programs[:5],
            projects=projects[:5]
        )

    @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_time_facets")
    def _extract_time_facets(self, smart_chunk: Any) -> TimeFacet:
        """Extract temporal information from SPC temporal_dynamics."""
        years = []
        periods = []

        if hasattr(smart_chunk, 'temporal_dynamics') and smart_chunk.temporal_dynamics:
            temp = smart_chunk.temporal_dynamics
            # Extract years from temporal_markers
            if hasattr(temp, 'temporal_markers'):
                for marker in temp.temporal_markers[:10]:
                    # marker format: (text, marker_type, position)
                    marker_text = marker[0] if isinstance(marker, (list, tuple)) else str(marker)
                    # Try to extract years (4-digit numbers between 2020-2030)
                    import re
                    year_matches = re.findall(r'\b(202[0-9]|203[0-9])\b', marker_text)
                    years.extend(int(y) for y in year_matches)

        # Also check strategic_context for temporal_horizon
        if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
            ctx = smart_chunk.strategic_context
            if hasattr(ctx, 'temporal_horizon'):
                periods.append(ctx.temporal_horizon)

        return TimeFacet(
            years=sorted(set(years))[:10],  # Unique and sorted
            periods=periods[:5]
```

```
3318:         )
3319:
3320:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_geo_facets")
3321:     def _extract_geo_facets(self, smart_chunk: Any) -> GeoFacet:
3322:         """Extract geographic information from SPC strategic_context."""
3323:         territories = []
3324:         regions = []
3325:
3326:         if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
3327:             ctx = smart_chunk.strategic_context
3328:             if hasattr(ctx, 'geographic_scope'):
3329:                 territories.append(ctx.geographic_scope)
3330:             # Could also parse from policy_entities with location types
3331:
3332:         return GeoFacet(
3333:             territories=territories[:5],
3334:             regions=regions[:5]
3335:         )
3336:
3337:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._build_provenance")
3338:     def _build_provenance(self, smart_chunk: Any) -> ProvenanceMap:
3339:         """Build provenance from SPC metadata."""
3340:         # Extract section info from section_hierarchy
3341:         source_section = None
3342:         if hasattr(smart_chunk, 'section_hierarchy') and smart_chunk.section_hierarchy:
3343:             source_section = " > ".join(smart_chunk.section_hierarchy[:3])
3344:
3345:         return ProvenanceMap(
3346:             source_page=None,  # SPC doesn't track page numbers
3347:             source_section=source_section,
3348:             extraction_method="smart_policy_chunking_v3.0"
3349:         )
3350:
3351:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_entities")
3352:     def _extract_entities(self, smart_chunk: Any) -> list[Entity]:
3353:         """Extract entities from SPC policy_entities."""
3354:         entities = []
3355:
3356:         if hasattr(smart_chunk, 'policy_entities') and smart_chunk.policy_entities:
3357:             for pe in smart_chunk.policy_entities[:20]:  # Limit to 20
3358:                 entity = Entity(
3359:                     text=pe.text if hasattr(pe, 'text') else str(pe),
3360:                     entity_type=pe.entity_type if hasattr(pe, 'entity_type') else 'unknown',
3361:                     confidence=pe.confidence if hasattr(pe, 'confidence') else ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_entities", "auto_param_L373_79", 0.8)
3362:                 )
3363:                 entities.append(entity)
3364:
3365:         return entities
3366:
3367:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_budget")
3368:     def _extract_budget(self, smart_chunk: Any) -> Budget | None:
3369:         """
3370:         Extract budget with comprehensive error handling and logging (H1.4).
3371:
3372:         Implements 4 regex patterns and robust year extraction with 4 fallback strategies.
```

```
3373:         """
3374:         if not (hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context):
3375:             return None
3376:
3377:         ctx = smart_chunk.strategic_context
3378:         if not (hasattr(ctx, 'budget_linkage') and ctx.budget_linkage):
3379:             return None
3380:
3381:         import re
3382:         budget_text = ctx.budget_linkage
3383:
3384:         # H1.4: 4 regex patterns for robust budget extraction
3385:         patterns = [
3386:             # Pattern 1: Currency symbol with optional scale
3387:             r'[\$]?\s*([0-9,]+(?:\.[0-9]+)?)\s*(millones|mil millones|billion|billones)?',
3388:             # Pattern 2: "X millones de pesos"
3389:             r'(\d+(?:\.\d+)?)\s*millones?\s+de\s+pesos',
3390:             # Pattern 3: COP currency code
3391:             r'COP\s*\$?\s*([0-9,]+(?:\.[0-9]+)?)',
3392:             # Pattern 4: "presupuesto de $X"
3393:             r'presupuesto\s+de\s+\$?\s*([0-9,]+(?:\.[0-9]+)?)',
3394:         ]
3395:
3396:         amount = None
3397:         scale_multiplier = 1
3398:
3399:         for pattern in patterns:
3400:             match = re.search(pattern, budget_text, re.IGNORECASE)
3401:             if match:
3402:                 try:
3403:                     amount_str = match.group(1).replace(',', '')
3404:                     amount = float(amount_str)
3405:
3406:                     # Detect scale from second group or text
3407:                     scale_text = budget_text.lower()
3408:                     if 'millones' in scale_text or 'million' in scale_text:
3409:                         scale_multiplier = 1_000_000
3410:                     elif 'mil millones' in scale_text or 'billion' in scale_text or 'billones' in scale_text:
3411:                         scale_multiplier = 1_000_000_000
3412:
3413:                     amount *= scale_multiplier
3414:                     break  # Stop at first match
3415:
3416:                 except (ValueError, IndexError) as e:
3417:                     self.logger.warning(f"Budget pattern matched but parsing failed: {e}")
3418:                     continue
3419:
3420:         if amount is None:
3421:             return None
3422:
3423:         # H1.4: Extract year with 4 fallback strategies
3424:         year = self._extract_budget_year(budget_text, smart_chunk)
3425:
3426:         # Build Budget object
3427:         use = ctx.policy_intent if hasattr(ctx, 'policy_intent') else "General"
3428:
```

```
3429:         self.logger.debug(
3430:             f"Extracted budget: amount={amount:,.2f}, year={year}, "
3431:             f"source='Strategic Context', use='{use[:30]}'"
3432:         )
3433:
3434:         return Budget(
3435:             source="Strategic Context",
3436:             use=use,
3437:             amount=amount,
3438:             year=year,
3439:             currency="COP"
3440:         )
3441:
3442:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_budget_year")
3443:     def _extract_budget_year(self, budget_text: str, smart_chunk: Any) -> int:
3444:         """
3445:         Extract budget year with 4 fallback strategies (H1.4).
3446:
3447:         Strategy 1: Extract from budget_text itself
3448:         Strategy 2: Extract from temporal_dynamics markers
3449:         Strategy 3: Use temporal_horizon from strategic_context
3450:         Strategy 4: Default to 2024
3451:         """
3452:         import re
3453:
3454:         # Strategy 1: Look for year in budget_text
3455:         year_match = re.search(r'\b(202[0-9]|203[0-9])\b', budget_text)
3456:         if year_match:
3457:             return int(year_match.group(1))
3458:
3459:         # Strategy 2: Check temporal_dynamics markers
3460:         if hasattr(smart_chunk, 'temporal_dynamics') and smart_chunk.temporal_dynamics:
3461:             temp = smart_chunk.temporal_dynamics
3462:             if hasattr(temp, 'temporal_markers') and temp.temporal_markers:
3463:                 for marker in temp.temporal_markers:
3464:                     marker_text = marker[0] if isinstance(marker, (list, tuple)) else str(marker)
3465:                     year_match = re.search(r'\b(202[0-9]|203[0-9])\b', marker_text)
3466:                     if year_match:
3467:                         return int(year_match.group(1))
3468:
3469:         # Strategy 3: Check strategic_context temporal_horizon
3470:         if hasattr(smart_chunk, 'strategic_context') and smart_chunk.strategic_context:
3471:             ctx = smart_chunk.strategic_context
3472:             if hasattr(ctx, 'temporal_horizon') and ctx.temporal_horizon:
3473:                 horizon_match = re.search(r'\b(202[0-9]|203[0-9])\b', ctx.temporal_horizon)
3474:                 if horizon_match:
3475:                     return int(horizon_match.group(1))
3476:
3477:         # Strategy 4: Default to 2024
3478:         self.logger.debug("No year found in budget context, defaulting to 2024")
3479:         return 2024
3480:
3481:     @calibrated_method("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_kpi")
3482:     def _extract_kpi(self, smart_chunk: Any) -> KPI | None:
3483:         """Extract KPI if chunk contains indicator information."""
3484:         # Check if chunk_type suggests this is a metric
```

```
3485:         chunk_type_str = smart_chunk.chunk_type.value if hasattr(smart_chunk.chunk_type, 'value') else str(smart_chunk.chunk_type)
3486:
3487:         if 'METRICA' in chunk_type_str.upper() or 'EVALUACION' in chunk_type_str.upper():
3488:             # Try to extract indicator from text
3489:             import re
3490:             text = smart_chunk.text
3491:             # Look for percentage targets (e.g., "90%", "meta del 85%")
3492:             target_match = re.search(r'meta.*?([0-9]+(?:\.[0-9]+)?)\s*%', text, re.IGNORECASE)
3493:             if target_match:
3494:                 return KPI(
3495:                     indicator_name=smart_chunk.text[:80],  # Use chunk text as indicator name
3496:                     target_value=float(target_match.group(1)),
3497:                     unit="%",
3498:                     year=None
3499:                 )
3500:
3501:         return None
3502:
3503:     def _calculate_quality_metrics(
3504:         self,
3505:         smart_chunks: list[Any],
3506:         chunk_graph: ChunkGraph
3507:     ) -> QualityMetrics:
3508:         """Calculate quality metrics from SPC data."""
3509:         # Provenance completeness
3510:         chunks_with_provenance = sum(
3511:             1 for c in chunk_graph.chunks.values()
3512:             if c.provenance and c.provenance.source_section
3513:         )
3514:         provenance_completeness = chunks_with_provenance / len(chunk_graph.chunks) if chunk_graph.chunks else ParameterLoaderV2.get("farfan_core.processing.
spc_ingestion.converter.SmartChunkConverter._extract_kpi", "auto_param_L526_110", 0.0)
3515:
3516:         # Average coherence from SPC coherence_score
3517:         avg_coherence = sum(sc.coherence_score for sc in smart_chunks) / len(smart_chunks) if smart_chunks else ParameterLoaderV2.get("farfan_core.processin
g.spc_ingestion.converter.SmartChunkConverter._extract_kpi", "auto_param_L529_112", 0.0)
3518:
3519:         # Average completeness from SPC completeness_index
3520:         avg_completeness = sum(sc.completeness_index for sc in smart_chunks) / len(smart_chunks) if smart_chunks else ParameterLoaderV2.get("farfan_core.pro
cessing.spc_ingestion.converter.SmartChunkConverter._extract_kpi", "auto_param_L532_118", 0.0)
3521:
3522:         # Budget consistency
3523:         chunks_with_budget = sum(1 for c in chunk_graph.chunks.values() if c.budget)
3524:         budget_consistency = chunks_with_budget / len(chunk_graph.chunks) if chunk_graph.chunks else ParameterLoaderV2.get("farfan_core.processing.spc_inges
tion.converter.SmartChunkConverter._extract_kpi", "auto_param_L536_101", 0.0)
3525:
3526:         # Temporal robustness
3527:         chunks_with_time = sum(1 for c in chunk_graph.chunks.values() if c.time_facets.years)
3528:         temporal_robustness = chunks_with_time / len(chunk_graph.chunks) if chunk_graph.chunks else ParameterLoaderV2.get("farfan_core.processing.spc_ingest
ion.converter.SmartChunkConverter._extract_kpi", "auto_param_L540_100", 0.0)
3529:
3530:         # Chunk context coverage (from edges)
3531:         chunks_with_edges = len({e[0] for e in chunk_graph.edges} | {e[1] for e in chunk_graph.edges})
3532:         chunk_context_coverage = chunks_with_edges / len(chunk_graph.chunks) if chunk_graph.chunks else ParameterLoaderV2.get("farfan_core.processing.spc_in
gestion.converter.SmartChunkConverter._extract_kpi", "auto_param_L544_104", 0.0)
3533:
3534:         return QualityMetrics(
```

```
3535:             provenance_completeness=provenance_completeness,
3536:             structural_consistency=avg_coherence,
3537:             boundary_f1=avg_completeness,
3538:             kpi_linkage_rate=ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.converter.SmartChunkConverter._extract_kpi", "auto_param_L550_29",
0.0),  # Would need KPI analysis
3539:             budget_consistency_score=budget_consistency,
3540:             temporal_robustness=temporal_robustness,
3541:             chunk_context_coverage=chunk_context_coverage
3542:         )
3543:
3544:     def _generate_integrity_index(
3545:         self,
3546:         chunk_hashes: dict[str, str],
3547:         document_metadata: dict[str, Any]
3548:     ) -> IntegrityIndex:
3549:         """
3550:         Generate cryptographic integrity index.
3551:
3552:         Uses BLAKE2b-256 to compute aggregate hash of all chunk hashes.
3553:         NOT a true Merkle tree - simply hashes sorted JSON representation.
3554:         """
3555:         # Generate root hash from all chunk hashes (sorted for determinism)
3556:         combined = json.dumps(chunk_hashes, sort_keys=True).encode('utf-8')
3557:         blake2b_root = hashlib.blake2b(combined, digest_size=32).hexdigest()
3558:
3559:         return IntegrityIndex(
3560:             blake2b_root=blake2b_root,
3561:             chunk_hashes=chunk_hashes
3562:         )
3563:
3564:     def _preserve_spc_rich_data(
3565:         self,
3566:         smart_chunks: list[Any],
3567:         document_metadata: dict[str, Any]
3568:     ) -> dict[str, Any]:
3569:         """
3570:         Preserve SPC rich data in metadata for executor access.
3571:
3572:         This is critical for enabling executors to access the full SPC analysis:
3573:         - Embeddings (semantic, policy, causal, temporal)
3574:         - Causal chains with evidence
3575:         - Strategic context
3576:         - Quality scores
3577:         """
3578:         enriched = dict(document_metadata)
3579:
3580:         # Store serializable SPC data
3581:         spc_rich_data = {}
3582:
3583:         for sc in smart_chunks:
3584:             chunk_data = {
3585:                 'chunk_id': sc.chunk_id,
3586:                 'semantic_density': sc.semantic_density,
3587:                 'coherence_score': sc.coherence_score,
3588:                 'completeness_index': sc.completeness_index,
3589:                 'strategic_importance': sc.strategic_importance,
```

```
3590:                 'information_density': sc.information_density,
3591:                 'actionability_score': sc.actionability_score,
3592:             }
3593:
3594:             # Add embeddings if available (as lists for JSON serialization)
3595:             # CRITICAL: Fail-fast if embeddings cannot be preserved (no silent data loss)
3596:             if hasattr(sc, 'semantic_embedding') and sc.semantic_embedding is not None:
3597:                 try:
3598:                     import numpy as np
3599:                 except ImportError as e:
3600:                     self.logger.error(
3601:                         f"Chunk {sc.chunk_id}: NumPy is required for embedding preservation but not available"
3602:                     )
3603:                     raise RuntimeError(
3604:                         "NumPy is required for SPC embedding preservation. "
3605:                         "Install with: pip install numpy>=1.26.0"
3606:                     ) from e
3607:
3608:                 # Validate embedding type
3609:                 if not isinstance(sc.semantic_embedding, np.ndarray):
3610:                     self.logger.error(
3611:                         f"Chunk {sc.chunk_id}: semantic_embedding is not np.ndarray, "
3612:                         f"got {type(sc.semantic_embedding)}"
3613:                     )
3614:                     raise TypeError(
3615:                         f"Expected semantic_embedding to be np.ndarray, got {type(sc.semantic_embedding)}"
3616:                     )
3617:
3618:                 # Convert to list for JSON serialization
3619:                 try:
3620:                     chunk_data['semantic_embedding'] = sc.semantic_embedding.tolist()
3621:                     chunk_data['embedding_dim'] = sc.semantic_embedding.shape[0]
3622:                     self.logger.debug(
3623:                         f"Chunk {sc.chunk_id}: Preserved embedding with dimension {sc.semantic_embedding.shape[0]}"
3624:                     )
3625:                 except (AttributeError, IndexError) as e:
3626:                     self.logger.error(
3627:                         f"Chunk {sc.chunk_id}: Failed to convert embedding to list: {e}"
3628:                     )
3629:                     raise RuntimeError(
3630:                         f"Embedding conversion failed for chunk {sc.chunk_id}: {e}"
3631:                     ) from e
3632:
3633:             # Add causal chain summary
3634:             if hasattr(sc, 'causal_chain') and sc.causal_chain:
3635:                 chunk_data['causal_chain_count'] = len(sc.causal_chain)
3636:                 chunk_data['causal_evidence'] = [
3637:                     {
3638:                         'dimension': ce.dimension if hasattr(ce, 'dimension') else 'unknown',
3639:                         'confidence': ce.confidence if hasattr(ce, 'confidence') else ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.converter.
SmartChunkConverter._extract_kpi", "auto_param_L651_86", 0.0),
3640:                     }
3641:                     for ce in sc.causal_chain[:5]  # Top 5
3642:                 ]
3643:
3644:             # Add strategic context summary
```

```
            if hasattr(sc, 'strategic_context') and sc.strategic_context:
                ctx = sc.strategic_context
                chunk_data['strategic_context'] = {
                    'policy_intent': ctx.policy_intent if hasattr(ctx, 'policy_intent') else None,
                    'implementation_phase': ctx.implementation_phase if hasattr(ctx, 'implementation_phase') else None,
                    'temporal_horizon': ctx.temporal_horizon if hasattr(ctx, 'temporal_horizon') else None,
                }

            # Add topic distribution
            if hasattr(sc, 'topic_distribution') and sc.topic_distribution:
                chunk_data['topic_distribution'] = dict(sc.topic_distribution)

            spc_rich_data[sc.chunk_id] = chunk_data

        enriched['spc_rich_data'] = spc_rich_data
        enriched['spc_version'] = 'SMART-CHUNK-3.0-FINAL'
        enriched['conversion_timestamp'] = document_metadata.get('processing_timestamp', 'unknown')

        self.logger.info(f"Preserved rich SPC data for {len(spc_rich_data)} chunks")

        return enriched


================================================================================
FILE: src/farfan_pipeline/processing/embedding_policy.py
================================================================================

"""
INTERNAL SPC COMPONENT

â\232 ï¸\217  USAGE RESTRICTION â\232 ï¸\217
============================================================================
This module implements SOTA semantic embedding and policy analysis for Smart
Policy Chunks. It MUST NOT be used as a standalone ingestion pipeline in the
canonical FARFAN flow.

Canonical entrypoint is scripts/run_policy_pipeline_verified.py.

This module is an INTERNAL COMPONENT of:
    src/farfan_core/processing/spc_ingestion.py (StrategicChunkingSystem)

DO NOT use this module directly as an independent pipeline. It is consumed
internally by the SPC core and should only be imported from within:
    - farfan_core.processing.spc_ingestion
    - Unit tests for SPC components

State-of-the-Art Components:
- BGE-M3 multilingual embeddings (2024 SOTA)
- Cross-encoder reranking for Spanish policy documents
- Bayesian uncertainty quantification for numerical analysis
- Graph-based multi-hop reasoning
============================================================================
"""

from __future__ import annotations
```

```
3701:
3702: import hashlib
3703: import logging
3704: import re
3705: from dataclasses import dataclass
3706: from enum import Enum
3707: from functools import lru_cache
3708: from typing import TYPE_CHECKING, Any, Literal, Protocol, TypedDict
3709:
3710: import numpy as np
3711: from sentence_transformers import CrossEncoder, SentenceTransformer
3712: from sklearn.metrics.pairwise import cosine_similarity
3713: from farfan_pipeline.core.parameters import ParameterLoaderV2
3714: from farfan_pipeline.core.calibration.decorators import calibrated_method
3715:
3716: if TYPE_CHECKING:
3717:     from collections.abc import Iterable
3718:
3719:     from numpy.typing import NDArray
3720:
3721: # ============================================================================
3722: # DESIGN CONSTANTS - Model Configuration
3723: # ============================================================================
3724:
3725: # Model constants
3726: DEFAULT_CROSS_ENCODER_MODEL = "cross-encoder/ms-marco-MiniLM-L-6-v2"
3727: MODEL_PARAPHRASE_MULTILINGUAL = "sentence-transformers/paraphrase-multilingual-mpnet-base-v2"
3728:
3729: # ============================================================================
3730: # TYPE SYSTEM - Python 3.10+ Type Safety
3731: # ============================================================================
3732:
3733: class PolicyDomain(Enum):
3734:     """
3735:     Colombian PDM policy areas (PA01-PA10) per canonical notation.
3736:
3737:     Values are loaded from questionnaire_monolith.json canonical_notation.
3738:     Use CanonicalPolicyArea from farfan_core.core.canonical_notation for dynamic access.
3739:     """
3740:
3741:     # Legacy IDs mapped to canonical codes for backward compatibility
3742:     P1 = "PA01"  # Derechos de las mujeres e igualdad de gÃ©nero
3743:     P2 = "PA02"  # PrevenciÃ³n de la violencia y protecciÃ³n frente al conflicto
3744:     P3 = "PA03"  # Ambiente sano, cambio climÃ¡tico, prevenciÃ³n y atenciÃ³n a desastres
3745:     P4 = "PA04"  # Derechos econÃ³micos, sociales y culturales
3746:     P5 = "PA05"  # Derechos de las vÃctimas y construcciÃ³n de paz
3747:     P6 = "PA06"  # Derecho al buen futuro de la niÃ±ez, adolescencia, juventud
3748:     P7 = "PA07"  # Tierras y territorios
3749:     P8 = "PA08"  # LÃderes y defensores de derechos humanos
3750:     P9 = "PA09"  # Crisis de derechos de personas privadas de la libertad
3751:     P10 = "PA10"  # MigraciÃ³n transfronteriza
3752:
3753: class AnalyticalDimension(Enum):
3754:     """
3755:     Analytical dimensions (D1-D6) per canonical notation.
3756:
```

```
3757:     Values reference canonical notation from questionnaire_monolith.json.
3758:     Use CanonicalDimension from farfan_core.core.canonical_notation for dynamic access.
3759:     """
3760:
3761:     D1 = "DIM01"  # INSUMOS - DiagnÃ³stico y Recursos
3762:     D2 = "DIM02"  # ACTIVIDADES - DiseÃ±o de IntervenciÃ³n
3763:     D3 = "DIM03"  # PRODUCTOS - Productos y Outputs
3764:     D4 = "DIM04"  # RESULTADOS - Resultados y Outcomes
3765:     D5 = "DIM05"  # IMPACTOS - Impactos de Largo Plazo
3766:     D6 = "DIM06"  # CAUSALIDAD - TeorÃa de Cambio
3767:
3768: class PDQIdentifier(TypedDict):
3769:     """Canonical P-D-Q identifier structure."""
3770:
3771:     question_unique_id: str  # P#-D#-Q#
3772:     policy: str  # P#
3773:     dimension: str  # D#
3774:     question: int  # Q#
3775:     rubric_key: str  # D#-Q#
3776:
3777: class PosteriorSampleRecord(TypedDict):
3778:     """Serializable posterior sample used by downstream Bayesian consumers."""
3779:
3780:     coherence: float
3781:
3782: class SemanticChunk(TypedDict):
3783:     """Structured semantic chunk with metadata."""
3784:
3785:     chunk_id: str
3786:     content: str
3787:     embedding: NDArray[np.float32]
3788:     metadata: dict[str, Any]
3789:     pdq_context: PDQIdentifier | None
3790:     token_count: int
3791:     position: tuple[int, int]  # (start, end) in document
3792:
3793: class PosteriorSample(TypedDict):
3794:     """Serialized posterior sample representation."""
3795:
3796:     coherence: float
3797:
3798: class BayesianEvaluation(TypedDict):
3799:     """Bayesian uncertainty-aware evaluation result."""
3800:
3801:     point_estimate: float  # 0.0-1.0
3802:     credible_interval_95: tuple[float, float]
3803:     posterior_samples: list[PosteriorSample]
3804:     evidence_strength: Literal["weak", "moderate", "strong", "very_strong"]
3805:     numerical_coherence: float  # Statistical consistency score
3806:     posterior_records: list[PosteriorSampleRecord]
3807:
3808: class EmbeddingProtocol(Protocol):
3809:     """Protocol for embedding models."""
3810:
3811:     def encode(
3812:         self, texts: list[str], batch_size: int = 32, normalize: bool = True
```

```
    ) -> NDArray[np.float32]: ...

def to_dict_samples(samples: NDArray[np.float32] | Iterable[float]) -> list[PosteriorSample]:
    """Convert posterior samples to the serialized TypedDict format."""

    array = np.asarray(list(samples) if not hasattr(samples, "shape") else samples, dtype=np.float32)
    flat = array.ravel()
    return [{"coherence": float(value)} for value in flat]

def samples_to_array(samples: NDArray[np.float32] | Iterable[PosteriorSample]) -> NDArray[np.float32]:
    """Normalize posterior samples into a numpy array for computation."""

    if isinstance(samples, np.ndarray):
        return samples.astype(np.float32)
    return np.array([sample["coherence"] for sample in samples], dtype=np.float32)

def ensure_content_schema(chunk: dict[str, Any]) -> dict[str, Any]:
    """Ensure chunk dictionaries expose the ``content`` key."""

    if "content" not in chunk and "text" in chunk:
        upgraded = dict(chunk)
        upgraded["content"] = upgraded.pop("text")
        return upgraded
    return chunk

# ============================================================================
# ADVANCED SEMANTIC CHUNKING - State-of-the-Art
# ============================================================================

@dataclass
class ChunkingConfig:
    """Configuration for semantic chunking optimized for PDM documents."""

    chunk_size: int = 512  # Tokens, optimized for policy documents
    chunk_overlap: int = 128  # Preserve context across chunks
    min_chunk_size: int = 64  # Avoid tiny fragments
    respect_boundaries: bool = True  # Sentence/paragraph boundaries
    preserve_tables: bool = True  # Keep tables intact
    detect_lists: bool = True  # Recognize enumerations
    section_aware: bool = True  # Understand document structure

class AdvancedSemanticChunker:
    """
    State-of-the-art semantic chunking for Colombian policy documents.

    Implements:
    - Recursive character splitting with semantic boundary preservation
    - Table structure detection and preservation
    - List and enumeration recognition
    - Hierarchical section awareness (P-D-Q structure)
    - Token-aware splitting (not just character-based)
    """

    # Colombian policy document patterns
    SECTION_HEADERS = re.compile(
        r"^(?:CAPÃ\215TULO|SECCIÃ\223N|ARTÃ\215CULO|PROGRAMA|PROYECTO|EJE)\s+[IVX\d]+",
```

```
3869:         re.MULTILINE | re.IGNORECASE,
3870:     )
3871:     TABLE_MARKERS = re.compile(r"(?:Tabla|Cuadro|Figura)\s+\d+", re.IGNORECASE)
3872:     LIST_MARKERS = re.compile(r"^[\s]*[â\200¢\-\*\d]+[\.\)]\s+", re.MULTILINE)
3873:     NUMERIC_INDICATORS = re.compile(
3874:         r"\b\d+(?:[.,]\d+)?(?:\s*%|millones?|mil|billones?)?\b", re.IGNORECASE
3875:     )
3876:
3877:     def __init__(self, config: ChunkingConfig) -> None:
3878:         self.config = config
3879:         self._logger = logging.getLogger(self.__class__.__name__)
3880:
3881:     def chunk_document(
3882:         self,
3883:         *,
3884:         text: str,
3885:         document_metadata: dict[str, Any],
3886:     ) -> list[SemanticChunk]:
3887:         """
3888:         Chunk document with advanced semantic awareness (keyword-only params).
3889:
3890:         Args:
3891:             text: Document text to chunk
3892:             document_metadata: Metadata dict with at least 'doc_id' key
3893:
3894:         Returns:
3895:             List of semantic chunks with preserved structure and P-D-Q context
3896:
3897:         Raises:
3898:             TypeError: If text is not a string
3899:             KeyError: If document_metadata missing required keys
3900:         """
3901:         # Runtime validation at ingress
3902:         if not isinstance(text, str):
3903:             raise TypeError(
3904:                 f"ERR_CONTRACT_MISMATCH[fn=chunk_document, param='text', "
3905:                 f"expected=str, got={type(text).__name__}]"
3906:             )
3907:
3908:         if not isinstance(document_metadata, dict):
3909:             raise TypeError(
3910:                 f"ERR_CONTRACT_MISMATCH[fn=chunk_document, param='document_metadata', "
3911:                 f"expected=dict, got={type(document_metadata).__name__}]"
3912:             )
3913:         # Preprocess: normalize whitespace, preserve structure
3914:         normalized_text = self._normalize_text(text)
3915:
3916:         # Extract structural elements
3917:         sections = self._extract_sections(normalized_text)
3918:         tables = self._extract_tables(normalized_text)
3919:         lists = self._extract_lists(normalized_text)
3920:
3921:         # Generate chunks with boundary preservation
3922:         raw_chunks = self._recursive_split(
3923:             normalized_text,
3924:             target_size=self.config.chunk_size,
```

```
3925:             overlap=self.config.chunk_overlap,
3926:         )
3927:
3928:         # Enrich chunks with metadata and P-D-Q context
3929:         semantic_chunks: list[SemanticChunk] = []
3930:
3931:         for idx, chunk_text in enumerate(raw_chunks):
3932:             # Infer P-D-Q context from chunk text
3933:             pdq_context = self._infer_pdq_context(chunk_text)
3934:
3935:             # Count tokens (approximation: Spanish has ~1.3 chars/token)
3936:             AVG_CHARS_PER_TOKEN = 1.3  # Source: Spanish language statistics
3937:             token_count = int(
3938:                 len(chunk_text) / AVG_CHARS_PER_TOKEN
3939:             )  # Approximate token count
3940:
3941:             # Create structured chunk
3942:             chunk_id = hashlib.sha256(
3943:                 f"{document_metadata.get('doc_id', '')}_{idx}_{chunk_text[:50]}".encode()
3944:             ).hexdigest()[:16]
3945:
3946:             semantic_chunk: SemanticChunk = {
3947:                 "chunk_id": chunk_id,
3948:                 "content": chunk_text,
3949:                 "embedding": np.array([]),  # Filled later
3950:                 "metadata": {
3951:                     "document_id": document_metadata.get("doc_id"),
3952:                     "chunk_index": idx,
3953:                     "has_table": self._contains_table(chunk_text, tables),
3954:                     "has_list": self._contains_list(chunk_text, lists),
3955:                     "has_numbers": bool(self.NUMERIC_INDICATORS.search(chunk_text)),
3956:                     "section_title": self._find_section(chunk_text, sections),
3957:                 },
3958:                 "pdq_context": pdq_context,
3959:                 "token_count": token_count,
3960:                 "position": (0, len(chunk_text)),  # Updated during splitting
3961:             }
3962:
3963:             semantic_chunks.append(ensure_content_schema(semantic_chunk))
3964:
3965:         self._logger.info(
3966:             "Created %d semantic chunks from document %s",
3967:             len(semantic_chunks),
3968:             document_metadata.get("doc_id", "unknown"),
3969:         )
3970:
3971:         return semantic_chunks
3972:
3973:     @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._normalize_text")
3974:     def _normalize_text(self, text: str) -> str:
3975:         """Normalize text while preserving structure."""
3976:         # Remove excessive whitespace but preserve paragraph breaks
3977:         text = re.sub(r"[ \t]+", " ", text)
3978:         text = re.sub(r"\n{3,}", "\n\n", text)
3979:         return text.strip()
3980:
```

```
    @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._recursive_split")
    def _recursive_split(self, text: str, target_size: int, overlap: int) -> list[str]:
        """
        Recursive character splitting with semantic boundary respect.

        Priority: Paragraph > Sentence > Word > Character
        """
        if len(text) <= target_size:
            return [text]

        chunks = []
        current_pos = 0

        while current_pos < len(text):
            # Calculate chunk end position
            end_pos = min(current_pos + target_size, len(text))

            # Try to find semantic boundary
            if end_pos < len(text):
                # Priority 1: Paragraph break
                paragraph_break = text.rfind("\n\n", current_pos, end_pos)
                if paragraph_break != -1 and paragraph_break > current_pos:
                    end_pos = paragraph_break + 2

                # Priority 2: Sentence boundary
                elif sentence_end := self._find_sentence_boundary(
                    text, current_pos, end_pos
                ):
                    end_pos = sentence_end

            chunk = text[current_pos:end_pos].strip()
            if len(chunk) >= self.config.min_chunk_size:
                chunks.append(chunk)

            # Move position with overlap
            current_pos = end_pos - overlap if overlap > 0 else end_pos

            # Prevent infinite loop
            if current_pos <= end_pos - target_size:
                current_pos = end_pos

        return chunks

    @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._find_sentence_boundary")
    def _find_sentence_boundary(self, text: str, start: int, end: int) -> int | None:
        """Find sentence boundary using Spanish punctuation rules."""
        # Spanish sentence endings: . ! ? ; followed by space or newline
        sentence_pattern = re.compile(r"[.!?;]\s+")

        matches = list(sentence_pattern.finditer(text, start, end))
        if matches:
            # Return position after punctuation and space
            return matches[-1].end()
        return None

    @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._extract_sections")
```

```
    def _extract_sections(self, text: str) -> list[dict[str, Any]]:
        """Extract document sections with hierarchical structure."""
        sections = []
        for match in self.SECTION_HEADERS.finditer(text):
            sections.append(
                {
                    "title": match.group(0),
                    "position": match.start(),
                    "end": match.end(),
                }
            )
        return sections

    # Number of characters to consider as table extent after marker
    TABLE_EXTENT_CHARS = 300

    @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._extract_tables")
    def _extract_tables(self, text: str) -> list[dict[str, Any]]:
        """Identify table regions in document."""
        tables = []
        for match in self.TABLE_MARKERS.finditer(text):
            # Heuristic: table extends ~TABLE_EXTENT_CHARS chars after marker
            tables.append(
                {
                    "marker": match.group(0),
                    "start": match.start(),
                    "end": min(match.end() + self.TABLE_EXTENT_CHARS, len(text)),
                }
            )
        return tables

    @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._extract_lists")
    def _extract_lists(self, text: str) -> list[dict[str, Any]]:
        """Identify list structures."""
        lists = []
        for match in self.LIST_MARKERS.finditer(text):
            lists.append({"marker": match.group(0), "position": match.start()})
        return lists

    def _infer_pdq_context(
        self,
        chunk_text: str,
    ) -> PDQIdentifier | None:
        """
        Infer P-D-Q context from chunk content.

        Uses heuristics based on Colombian policy vocabulary.
        """
        # Policy-specific keywords (simplified for example)
        policy_keywords = {
            "PA01": ["mujer", "gÃ©nero", "igualdad", "equidad"],
            "PA02": ["violencia", "conflicto", "seguridad", "prevenciÃ³n"],
            "PA03": ["ambiente", "clima", "desastre", "riesgo"],
            "PA04": ["econÃ³mico", "social", "cultural", "empleo"],
            "PA05": ["vÃctima", "paz", "reconciliaciÃ³n", "reparaciÃ³n"],
            "PA06": ["niÃ±ez", "adolescente", "juventud", "futuro"],
```

```
4093:             "PA07": ["tierra", "territorio", "rural", "agrario"],
4094:             "PA08": ["lÃ-der", "defensor", "derechos humanos"],
4095:             "PA09": ["privado libertad", "cÃ¡rcel", "reclusiÃ³n"],
4096:             "PA10": ["migraciÃ³n", "frontera", "venezolano"],
4097:         }
4098:
4099:         dimension_keywords = {
4100:             "DIM01": ["diagnÃ³stico", "baseline", "situaciÃ³n", "recurso"],
4101:             "DIM02": ["diseÃ±o", "estrategia", "intervenciÃ³n", "actividad"],
4102:             "DIM03": ["producto", "output", "entregable", "meta"],
4103:             "DIM04": ["resultado", "outcome", "efecto", "cambio"],
4104:             "DIM05": ["impacto", "largo plazo", "sostenibilidad"],
4105:             "DIM06": ["teorÃ-a", "causal", "coherencia", "lÃ³gica"],
4106:         }
4107:
4108:         # Score policies and dimensions
4109:         policy_scores = {
4110:             policy: sum(1 for kw in keywords if kw.lower() in chunk_text.lower())
4111:             for policy, keywords in policy_keywords.items()
4112:         }
4113:
4114:         dimension_scores = {
4115:             dim: sum(1 for kw in keywords if kw.lower() in chunk_text.lower())
4116:             for dim, keywords in dimension_keywords.items()
4117:         }
4118:
4119:         # Select best match if confidence is sufficient
4120:         best_policy = max(policy_scores, key=policy_scores.get)
4121:         best_dimension = max(dimension_scores, key=dimension_scores.get)
4122:
4123:         if policy_scores[best_policy] > 0 and dimension_scores[best_dimension] > 0:
4124:             # Generate canonical identifier
4125:             question_num = 1  # Simplified; real system would infer from context
4126:             question_code = f"Q{question_num:03d}"
4127:
4128:             return PDQIdentifier(
4129:                 question_unique_id=f"{best_policy}-{best_dimension}-{question_code}",
4130:                 policy=best_policy,
4131:                 dimension=best_dimension,
4132:                 question=question_num,
4133:                 rubric_key=f"{best_dimension}-{question_code}",
4134:             )
4135:
4136:         return None
4137:
4138:     def _contains_table(
4139:         self, chunk_text: str, tables: list[dict[str, Any]]
4140:     ) -> bool:
4141:         """Check if chunk contains table markers."""
4142:         return any(
4143:             table["marker"] in chunk_text
4144:             for table in tables
4145:         )
4146:
4147:     @calibrated_method("farfan_core.processing.embedding_policy.AdvancedSemanticChunker._contains_list")
4148:     def _contains_list(self, chunk_text: str, lists: list[dict[str, Any]]) -> bool:
```

```
4149:         """Check if chunk contains list structures."""
4150:         return bool(self.LIST_MARKERS.search(chunk_text))
4151:
4152:     def _find_section(
4153:         self, chunk_text: str, sections: list[dict[str, Any]]
4154:     ) -> str | None:
4155:         """Find section title for chunk."""
4156:         # Simplified: would use position-based matching in production
4157:         for section in sections:
4158:             if section["title"][:20] in chunk_text:
4159:                 return section["title"]
4160:         return None
4161:
4162: # ==============================================================================
4163: # BAYESIAN NUMERICAL ANALYSIS - Rigorous Statistical Framework
4164: # ==============================================================================
4165:
4166: class BayesianNumericalAnalyzer:
4167:     """
4168:     Bayesian framework for uncertainty-aware numerical policy analysis.
4169:
4170:     Implements:
4171:     - Beta-Binomial conjugate prior for proportions
4172:     - Normal-Normal conjugate prior for continuous metrics
4173:     - Bayesian hypothesis testing for policy comparisons
4174:     - Credible interval estimation
4175:     - Evidence strength quantification (Bayes factors)
4176:     """
4177:
4178:     def __init__(self, prior_strength: float = 1.0) -> None:
4179:         """
4180:         Initialize Bayesian analyzer.
4181:
4182:         Args:
4183:             prior_strength: Prior belief strength (ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__init__", "auto
_param_L510_51", 1.0) = weak, 1ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__init__", "auto_param_L510_64", 0.0) = str
ong)
4184:         """
4185:         self.prior_strength = prior_strength
4186:         self._logger = logging.getLogger(self.__class__.__name__)
4187:         self._rng = np.random.default_rng()
4188:
4189:     def evaluate_policy_metric(
4190:         self,
4191:         observed_values: list[float],
4192:         n_posterior_samples: int = 10000,
4193:         **kwargs: Any
4194:     ) -> BayesianEvaluation:
4195:         """
4196:         Bayesian evaluation of policy metric with uncertainty quantification.
4197:
4198:         Returns posterior distribution, credible intervals, and evidence strength.
4199:
4200:         Args:
4201:             observed_values: List of observed metric values
4202:             n_posterior_samples: Number of posterior samples to generate
```

```
            **kwargs: Additional optional parameters for compatibility

        Returns:
            BayesianEvaluation with posterior samples and credible intervals
        """
        if not observed_values:
            return self._null_evaluation()

        obs_array = np.array(observed_values)

        # Choose likelihood model based on data characteristics
        if all(0 <= v <= 1 for v in observed_values):
            # Proportion/probability metric: use Beta-Binomial
            posterior_samples = self._beta_binomial_posterior(
                obs_array, n_posterior_samples
            )
        else:
            # Continuous metric: use Normal-Normal
            posterior_samples = self._normal_normal_posterior(
                obs_array, n_posterior_samples
            )

        # Compute statistics
        point_estimate = float(np.median(posterior_samples))
        ci_lower, ci_upper = (
            float(np.percentile(posterior_samples, 2.5)),
            float(np.percentile(posterior_samples, 97.5)),
        )

        # Quantify evidence strength using posterior width
        ci_width = ci_upper - ci_lower
        evidence_strength = self._classify_evidence_strength(ci_width)

        # Assess numerical coherence (consistency of observations)
        coherence = self._compute_coherence(obs_array)

        serialized_samples = to_dict_samples(posterior_samples)

        return BayesianEvaluation(
            point_estimate=point_estimate,
            credible_interval_95=(ci_lower, ci_upper),
            posterior_samples=serialized_samples,
            evidence_strength=evidence_strength,
            numerical_coherence=coherence,
            posterior_records=self.serialize_posterior_samples(posterior_samples),
        )

    def _beta_binomial_posterior(
        self, observations: NDArray[np.float32], n_samples: int
    ) -> NDArray[np.float32]:
        """
        Beta-Binomial conjugate posterior for proportion metrics.

        Prior: Beta(Î±, Î²)
        Likelihood: Binomial
        Posterior: Beta(Î± + successes, Î² + failures)
```

```
4259:         """
4260:         # Prior parameters (weakly informative)
4261:         alpha_prior = self.prior_strength
4262:         beta_prior = self.prior_strength
4263:
4264:         # Convert proportions to successes/failures
4265:         n_obs = len(observations)
4266:         sum_success = np.sum(observations)  # If already in [0,1]
4267:
4268:         # Posterior parameters
4269:         alpha_post = alpha_prior + sum_success
4270:         beta_post = beta_prior + (n_obs - sum_success)
4271:
4272:         # Sample from posterior
4273:         posterior_samples = self._rng.beta(alpha_post, beta_post, size=n_samples)
4274:
4275:         return posterior_samples.astype(np.float32)
4276:
4277:     def _normal_normal_posterior(
4278:         self, observations: NDArray[np.float32], n_samples: int
4279:     ) -> NDArray[np.float32]:
4280:         """
4281:         Normal-Normal conjugate posterior for continuous metrics.
4282:
4283:         Prior: Normal(Î¼â\202\200, Ï\203â\202\200Â²)
4284:         Likelihood: Normal(Î¼, Ï\203Â²)
4285:         Posterior: Normal(Î¼_post, Ï\203_postÂ²)
4286:         """
4287:         n_obs = len(observations)
4288:         obs_mean = np.mean(observations)
4289:         obs_std = np.std(observations, ddof=1) if n_obs > 1 else ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__
init__", "auto_param_L616_65", 1.0)
4290:
4291:         # Prior parameters (weakly informative centered on observed mean)
4292:         mu_prior = obs_mean
4293:         sigma_prior = obs_std * self.prior_strength
4294:
4295:         # Posterior parameters (conjugate update)
4296:         precision_prior = 1 / (sigma_prior**2)
4297:         precision_likelihood = n_obs / (obs_std**2)
4298:
4299:         precision_post = precision_prior + precision_likelihood
4300:         mu_post = (
4301:             precision_prior * mu_prior + precision_likelihood * obs_mean
4302:         ) / precision_post
4303:         sigma_post = np.sqrt(1 / precision_post)
4304:
4305:         # Sample from posterior
4306:         posterior_samples = self._rng.normal(mu_post, sigma_post, size=n_samples)
4307:
4308:         return posterior_samples.astype(np.float32)
4309:
4310:     def _classify_evidence_strength(
4311:         self, credible_interval_width: float, **kwargs: Any
4312:     ) -> Literal["weak", "moderate", "strong", "very_strong"]:
4313:         """Classify evidence strength based on posterior uncertainty.
```

```
 4314:
 4315:         Args:
 4316:             credible_interval_width: Width of the 95% credible interval
 4317:             **kwargs: Additional optional parameters for compatibility
 4318:
 4319:         Returns:
 4320:             Evidence strength classification (weak/moderate/strong/very_strong)
 4321:         """
 4322:         if credible_interval_width > ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__init__", "auto_param_L649_37
", 0.5):
 4323:             return "weak"
 4324:         elif credible_interval_width > ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__init__", "auto_param_L651_
39", 0.3):
 4325:             return "moderate"
 4326:         elif credible_interval_width > ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer.__init__", "auto_param_L653_
39", 0.15):
 4327:             return "strong"
 4328:         else:
 4329:             return "very_strong"
 4330:
 4331:     @calibrated_method("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._compute_coherence")
 4332:     def _compute_coherence(self, observations: NDArray[np.float32], **kwargs: Any) -> float:
 4333:         """
 4334:         Compute numerical coherence (consistency) score.
 4335:
 4336:         Uses coefficient of variation and statistical tests.
 4337:
 4338:         Args:
 4339:             observations: Array of observed values
 4340:             **kwargs: Additional optional parameters for compatibility
 4341:
 4342:         Returns:
 4343:             Coherence score in [0, 1]
 4344:         """
 4345:         if len(observations) < 2:
 4346:             return ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._compute_coherence", "auto_param_L673_19", 1.0)
 4347:
 4348:         # Coefficient of variation
 4349:         mean_val = np.mean(observations)
 4350:         std_val = np.std(observations, ddof=1)
 4351:
 4352:         if mean_val == 0:
 4353:             return ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._compute_coherence", "auto_param_L680_19", 0.0)
 4354:
 4355:         cv = std_val / abs(mean_val)
 4356:
 4357:         # Normalize: lower CV = higher coherence
 4358:         coherence = np.exp(-cv)  # Exponential decay
 4359:
 4360:         return float(np.clip(coherence, ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._compute_coherence", "auto_
param_L687_40", 0.0), ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._compute_coherence", "auto_param_L687_45", 1.0)))
 4361:
 4362:     @calibrated_method("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation")
 4363:     def _null_evaluation(self) -> BayesianEvaluation:
 4364:         """Return null evaluation when no data available."""
 4365:         null_samples = to_dict_samples(np.array([ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation",
```

```
 "auto_param_L692_49", 0.0)], dtype=np.float32))
 4366:
 4367:         return BayesianEvaluation(
 4368:             point_estimate=ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L695_27",
 0.0),
 4369:             credible_interval_95=(ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L6
96_34", 0.0), ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L696_39", 0.0)),
 4370:             posterior_samples=null_samples,
 4371:             evidence_strength="weak",
 4372:             numerical_coherence=ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L699
_32", 0.0),
 4373:             posterior_records=[{"coherence": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "au
to_param_L700_45", 0.0)}],
 4374:         )
 4375:
 4376:     def serialize_posterior_samples(
 4377:         self, samples: NDArray[np.float32]
 4378:     ) -> list[PosteriorSampleRecord]:
 4379:         """Convert posterior samples into standardized coherence records.
 4380:
 4381:         Safely handles None or non-array inputs and limits the number of
 4382:         serialized records to avoid excessive memory use.
 4383:         """
 4384:         if samples is None:
 4385:             return []
 4386:
 4387:         # Ensure a 1-D numpy array of floats
 4388:         arr = np.asarray(samples, dtype=np.float32).ravel()
 4389:
 4390:         # Prevent accidental excessive memory use when serializing huge arrays
 4391:         MAX_RECORDS = 10000
 4392:         values = arr.tolist()
 4393:         if len(values) > MAX_RECORDS:
 4394:             values = values[:MAX_RECORDS]
 4395:
 4396:         return [{"coherence": float(v)} for v in values]
 4397:
 4398:     def compare_policies(
 4399:         self,
 4400:         policy_a_values: list[float],
 4401:         policy_b_values: list[float],
 4402:     ) -> dict[str, Any]:
 4403:         """
 4404:         Bayesian comparison of two policy metrics.
 4405:
 4406:         Returns probability that A > B and Bayes factor.
 4407:         """
 4408:         if not policy_a_values or not policy_b_values:
 4409:             return {"probability_a_better": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "aut
o_param_L736_44", 0.5), "bayes_factor": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L73
6_65", 1.0)}
 4410:
 4411:         # Get posterior distributions
 4412:         eval_a = self.evaluate_policy_metric(policy_a_values)
 4413:         eval_b = self.evaluate_policy_metric(policy_b_values)
 4414:
```

```
4415:         # Compute probability that A > B and clip to avoid exact 0/1 which can cause
4416:         # division-by-zero in subsequent Bayes factor calculation
4417:         samples_a = samples_to_array(eval_a["posterior_samples"])
4418:         samples_b = samples_to_array(eval_b["posterior_samples"])
4419:
4420:         # Compute probability that A > B and clip to avoid exact 0/1 which can cause
4421:         # division-by-zero in subsequent Bayes factor calculation.
4422:         prob_a_better = float(np.mean(samples_a > samples_b))
4423:         prob_a_better = float(np.clip(prob_a_better, 1e-6, ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_ev
aluation", "auto_param_L750_59", 1.0) - 1e-6))
4424:
4425:         # Compute Bayes factor (simplified)
4426:         if prob_a_better > ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L753_27",
 0.5):
4427:             bayes_factor = prob_a_better / (1 - prob_a_better)
4428:         else:
4429:             bayes_factor = (1 - prob_a_better) / prob_a_better
4430:
4431:         return {
4432:             "probability_a_better": float(prob_a_better),
4433:             "bayes_factor": float(bayes_factor),
4434:             "difference_mean": float(np.mean(samples_a - samples_b)),
4435:             "difference_ci_95": (
4436:                 float(
4437:                     np.percentile(
4438:                         samples_a - samples_b,
4439:                         2.5,
4440:                     )
4441:                 ),
4442:                 float(
4443:                     np.percentile(
4444:                         samples_a - samples_b,
4445:                         97.5,
4446:                     )
4447:                 ),
4448:             ),
4449:         }
4450:
4451: # ============================================================================
4452: # CROSS-ENCODER RERANKING - State-of-the-Art Retrieval
4453: # ============================================================================
4454:
4455: class PolicyCrossEncoderReranker:
4456:     """
4457:     Cross-encoder reranking optimized for Spanish policy documents.
4458:
4459:     Uses transformer-based cross-attention for precise relevance scoring.
4460:     Superior to bi-encoder + cosine similarity for final ranking.
4461:     """
4462:
4463:     def __init__(
4464:         self,
4465:         model_name: str = DEFAULT_CROSS_ENCODER_MODEL,
4466:         max_length: int = 512,
4467:         retry_handler=None,
4468:     ) -> None:
```

```
4469:         """
4470:         Initialize cross-encoder reranker.
4471:
4472:         Args:
4473:             model_name: HuggingFace model name (multilingual preferred)
4474:             max_length: Maximum sequence length for cross-encoder
4475:             retry_handler: Optional RetryHandler for model loading
4476:
4477:         Raises:
4478:             RuntimeError: If online model download is required but HF_ONLINE=0
4479:         """
4480:         self._logger = logging.getLogger(self.__class__.__name__)
4481:         self.retry_handler = retry_handler
4482:
4483:         # Check dependency lockdown before attempting model load
4484:         from farfan_pipeline.core.dependency_lockdown import _is_model_cached, get_dependency_lockdown
4485:         lockdown = get_dependency_lockdown()
4486:
4487:         # Check if we're trying to download a remote model when offline
4488:         if not _is_model_cached(model_name):
4489:             lockdown.check_online_model_access(
4490:                 model_name=model_name,
4491:                 operation="load CrossEncoder model"
4492:             )
4493:
4494:         # Load model with retry logic if available
4495:         if retry_handler:
4496:             try:
4497:                 from retry_handler import DependencyType
4498:
4499:                 @retry_handler.with_retry(
4500:                     DependencyType.EMBEDDING_SERVICE,
4501:                     operation_name="load_cross_encoder",
4502:                     exceptions=(OSError, IOError, ConnectionError, RuntimeError)
4503:                 )
4504:                 def load_model():
4505:                     return CrossEncoder(model_name, max_length=max_length)
4506:
4507:                 self.model = load_model()
4508:                 self._logger.info(f"Cross-encoder loaded with retry protection: {model_name}")
4509:             except Exception as e:
4510:                 self._logger.error(f"Failed to load cross-encoder: {e}")
4511:                 raise
4512:         else:
4513:             self.model = CrossEncoder(model_name, max_length=max_length)
4514:             self._logger.info(f"Cross-encoder loaded: {model_name}")
4515:
4516:     def rerank(
4517:         self,
4518:         query: str,
4519:         candidates: list[SemanticChunk],
4520:         top_k: int = 10,
4521:         min_score: float = ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L848_27",
0.0),
4522:     ) -> list[tuple[SemanticChunk, float]]:
4523:         """
```

```
        Rerank candidates using cross-encoder attention.

        Returns top-k chunks with relevance scores.
        """
        if not candidates:
            return []

        # Prepare query-document pairs
        pairs = [(query, chunk["content"]) for chunk in candidates]

        # Score with cross-encoder
        scores = self.model.predict(pairs, show_progress_bar=False)

        # Combine chunks with scores and sort
        ranked = sorted(zip(candidates, scores, strict=False), key=lambda x: x[1], reverse=True)

        # Filter by minimum score and limit to top_k
        filtered = [
            (chunk, float(score)) for chunk, score in ranked if score >= min_score
        ][:top_k]

        self._logger.info(
            "Reranked %d candidates, returned %d with min_score=%.2f",
            len(candidates),
            len(filtered),
            min_score,
        )

        return filtered

# ============================================================================
# MAIN EMBEDDING SYSTEM - Orchestrator
# ============================================================================

@dataclass
class PolicyEmbeddingConfig:
    """Configuration for policy embedding system."""

    # Model selection
    embedding_model: str = MODEL_PARAPHRASE_MULTILINGUAL
    cross_encoder_model: str = DEFAULT_CROSS_ENCODER_MODEL

    # Chunking parameters
    chunk_size: int = 512
    chunk_overlap: int = 128

    # Retrieval parameters
    top_k_candidates: int = 50  # Bi-encoder retrieval
    top_k_rerank: int = 10  # Cross-encoder rerank
    mmr_lambda: float = ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L900_24", 0.
7)  # Diversity vs relevance trade-off

    # Bayesian analysis
    prior_strength: float = ParameterLoaderV2.get("farfan_core.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation", "auto_param_L903_28"
, 1.0)  # Weakly informative prior

```

```
    # Performance
    batch_size: int = 32
    normalize_embeddings: bool = True

class PolicyAnalysisEmbedder:
    """
    Production-ready embedding system for Colombian PDM analysis.

    Implements complete pipeline:
    1. Advanced semantic chunking with P-D-Q awareness
    2. Multilingual embedding (Spanish-optimized)
    3. Bi-encoder retrieval + cross-encoder reranking
    4. Bayesian numerical analysis with uncertainty quantification
    5. MMR-based diversification

    Thread-safe, production-grade, fully typed.
    """

    def __init__(self, config: PolicyEmbeddingConfig, retry_handler=None) -> None:
        self.config = config
        self._logger = logging.getLogger(self.__class__.__name__)
        self.retry_handler = retry_handler

        # Check dependency lockdown before attempting model loads
        from farfan_pipeline.core.dependency_lockdown import _is_model_cached, get_dependency_lockdown
        lockdown = get_dependency_lockdown()

        # Check if we're trying to download remote models when offline
        if not _is_model_cached(config.embedding_model):
            lockdown.check_online_model_access(
                model_name=config.embedding_model,
                operation="load SentenceTransformer embedding model"
            )

        # Initialize embedding model with retry logic
        if retry_handler:
            try:
                from retry_handler import DependencyType

                @retry_handler.with_retry(
                    DependencyType.EMBEDDING_SERVICE,
                    operation_name="load_sentence_transformer",
                    exceptions=(OSError, IOError, ConnectionError, RuntimeError)
                )
                def load_embedding_model():
                    return SentenceTransformer(config.embedding_model)

                self._logger.info("Initializing embedding model with retry: %s", config.embedding_model)
                self.embedding_model = load_embedding_model()
            except Exception as e:
                self._logger.error(f"Failed to load embedding model: {e}")
                raise
        else:
            self._logger.info("Initializing embedding model: %s", config.embedding_model)
            self.embedding_model = SentenceTransformer(config.embedding_model)

```

```
        # Initialize cross-encoder with retry logic
        self._logger.info("Initializing cross-encoder: %s", config.cross_encoder_model)
        self.cross_encoder = PolicyCrossEncoderReranker(
            config.cross_encoder_model,
            retry_handler=retry_handler
        )

        self.chunker = AdvancedSemanticChunker(
            ChunkingConfig(
                chunk_size=config.chunk_size,
                chunk_overlap=config.chunk_overlap,
            )
        )

        self.bayesian_analyzer = BayesianNumericalAnalyzer(
            prior_strength=config.prior_strength
        )

        # Cache
        self._embedding_cache: dict[str, NDArray[np.float32]] = {}
        self._chunk_cache: dict[str, list[SemanticChunk]] = {}

    def process_document(
        self,
        document_text: str,
        document_metadata: dict[str, Any],
    ) -> list[SemanticChunk]:
        """
        Process complete PDM document into semantic chunks with embeddings.

        Args:
            document_text: Full document text
            document_metadata: Metadata including doc_id, municipality, year

        Returns:
            List of semantic chunks with embeddings and P-D-Q context
        """
        doc_id = document_metadata.get("doc_id", "unknown")
        self._logger.info("Processing document: %s", doc_id)

        # Check cache
        if doc_id in self._chunk_cache:
            self._logger.info(
                "Retrieved %d chunks from cache", len(self._chunk_cache[doc_id])
            )
            return self._chunk_cache[doc_id]

        # Chunk document with semantic awareness
        chunks = self.chunker.chunk_document(document_text, document_metadata)

        # Generate embeddings in batches
        chunk_texts = [chunk["content"] for chunk in chunks]
        embeddings = self._embed_texts(chunk_texts)

        # Attach embeddings to chunks
        for chunk, embedding in zip(chunks, embeddings, strict=False):
```

```
            chunk["embedding"] = embedding

        # Cache results
        self._chunk_cache[doc_id] = chunks

        self._logger.info(
            "Processed document %s: %d chunks, avg tokens: %.1f",
            doc_id,
            len(chunks),
            np.mean([c["token_count"] for c in chunks]),
        )

        return chunks

    def semantic_search(
        self,
        query: str,
        document_chunks: list[SemanticChunk],
        pdq_filter: PDQIdentifier | None = None,
        use_reranking: bool = True,
    ) -> list[tuple[SemanticChunk, float]]:
        """
        Advanced semantic search with P-D-Q filtering and reranking.

        Pipeline:
        1. Bi-encoder retrieval (fast, approximate)
        2. P-D-Q filtering (if specified)
        3. Cross-encoder reranking (precise)
        4. MMR diversification

        Args:
            query: Search query
            document_chunks: Pool of chunks to search
            pdq_filter: Optional P-D-Q context filter
            use_reranking: Enable cross-encoder reranking

        Returns:
            Ranked list of (chunk, score) tuples
        """
        if not document_chunks:
            return []

        # Bi-encoder retrieval: fast approximate search
        chunk_embeddings = np.vstack([c["embedding"] for c in document_chunks])
        query_embedding = self._embed_texts([query])[0]
        similarities = cosine_similarity(
            query_embedding.reshape(1, -1), chunk_embeddings
        ).ravel()

        # Get top-k candidates
        top_indices = np.argsort(-similarities)[: self.config.top_k_candidates]
        candidates = [document_chunks[i] for i in top_indices]

        # Apply P-D-Q filter if specified
        if pdq_filter:
            candidates = self._filter_by_pdq(candidates, pdq_filter)
```

```
4746:             self._logger.info(
4747:                 "Filtered to %d chunks matching P-D-Q context", len(candidates)
4748:             )
4749:
4750:         if not candidates:
4751:             return []
4752:
4753:         # Cross-encoder reranking for precision
4754:         if use_reranking:
4755:             reranked = self.cross_encoder.rerank(
4756:                 query, candidates, top_k=self.config.top_k_rerank
4757:             )
4758:         else:
4759:             # Use bi-encoder scores
4760:             candidate_indices = [document_chunks.index(c) for c in candidates]
4761:             reranked = [
4762:                 (candidates[i], float(similarities[candidate_indices[i]]))
4763:                 for i in range(len(candidates))
4764:             ]
4765:             reranked.sort(key=lambda x: x[1], reverse=True)
4766:             reranked = reranked[: self.config.top_k_rerank]
4767:
4768:         # MMR diversification
4769:         if len(reranked) > 1:
4770:             reranked = self._apply_mmr(reranked)
4771:
4772:         return reranked
4773:
4774:     def evaluate_policy_numerical_consistency(
4775:         self,
4776:         chunks: list[SemanticChunk],
4777:         pdq_context: PDQIdentifier,
4778:     ) -> BayesianEvaluation:
4779:         """
4780:         Bayesian evaluation of numerical consistency for policy metric.
4781:
4782:         Extracts numerical values from chunks matching P-D-Q context,
4783:         performs rigorous statistical analysis with uncertainty quantification.
4784:
4785:         Args:
4786:             chunks: Document chunks to analyze
4787:             pdq_context: P-D-Q context to filter relevant chunks
4788:
4789:         Returns:
4790:             Bayesian evaluation with credible intervals and evidence strength
4791:         """
4792:         # Filter chunks by P-D-Q context
4793:         relevant_chunks = self._filter_by_pdq(chunks, pdq_context)
4794:
4795:         if not relevant_chunks:
4796:             self._logger.warning(
4797:                 "No chunks found for P-D-Q context: %s",
4798:                 pdq_context["question_unique_id"],
4799:             )
4800:             return self.bayesian_analyzer._null_evaluation()
4801:
```

```
4802:         # Extract numerical values from chunks
4803:         numerical_values = self._extract_numerical_values(relevant_chunks)
4804:
4805:         if not numerical_values:
4806:             self._logger.warning(
4807:                 "No numerical values extracted from %d chunks", len(relevant_chunks)
4808:             )
4809:             return self.bayesian_analyzer._null_evaluation()
4810:
4811:         # Perform Bayesian evaluation
4812:         evaluation = self.bayesian_analyzer.evaluate_policy_metric(numerical_values)
4813:
4814:         self._logger.info(
4815:             "Evaluated %d numerical values for %s: point_estimate=%.3f, CI=[%.3f, %.3f], evidence=%s",
4816:             len(numerical_values),
4817:             pdq_context["rubric_key"],
4818:             evaluation["point_estimate"],
4819:             evaluation["credible_interval_95"][0],
4820:             evaluation["credible_interval_95"][1],
4821:             evaluation["evidence_strength"],
4822:         )
4823:
4824:         return evaluation
4825:
4826:     def compare_policy_interventions(
4827:         self,
4828:         intervention_a_chunks: list[SemanticChunk],
4829:         intervention_b_chunks: list[SemanticChunk],
4830:         pdq_context: PDQIdentifier,
4831:     ) -> dict[str, Any]:
4832:         """
4833:         Bayesian comparison of two policy interventions.
4834:
4835:         Returns probability and evidence for superiority.
4836:         """
4837:         values_a = self._extract_numerical_values(
4838:             self._filter_by_pdq(intervention_a_chunks, pdq_context)
4839:         )
4840:         values_b = self._extract_numerical_values(
4841:             self._filter_by_pdq(intervention_b_chunks, pdq_context)
4842:         )
4843:
4844:         return self.bayesian_analyzer.compare_policies(values_a, values_b)
4845:
4846:     def generate_pdq_report(
4847:         self,
4848:         document_chunks: list[SemanticChunk],
4849:         target_pdq: PDQIdentifier,
4850:     ) -> dict[str, Any]:
4851:         """
4852:         Generate comprehensive analytical report for P-D-Q question.
4853:
4854:         Combines semantic search, numerical analysis, and evidence synthesis.
4855:         """
4856:         # Semantic search for relevant content
4857:         query = self._generate_query_from_pdq(target_pdq)
```

```
        relevant_chunks = self.semantic_search(
            query, document_chunks, pdq_filter=target_pdq
        )

        # Numerical consistency analysis
        numerical_eval = self.evaluate_policy_numerical_consistency(
            document_chunks, target_pdq
        )

        # Extract key evidence passages
        evidence_passages = [
            {
                "content": chunk["content"][:300],
                "relevance_score": float(score),
                "metadata": chunk["metadata"],
            }
            for chunk, score in relevant_chunks[:3]
        ]

        # Synthesize report
        report = {
            "question_unique_id": target_pdq["question_unique_id"],
            "rubric_key": target_pdq["rubric_key"],
            "evidence_count": len(relevant_chunks),
            "numerical_evaluation": {
                "point_estimate": numerical_eval["point_estimate"],
                "credible_interval_95": numerical_eval["credible_interval_95"],
                "evidence_strength": numerical_eval["evidence_strength"],
                "numerical_coherence": numerical_eval["numerical_coherence"],
            },
            "evidence_passages": evidence_passages,
            "confidence": self._compute_overall_confidence(
                relevant_chunks, numerical_eval
            ),
        }

        return report

    # ========================================================================
    # PRIVATE METHODS
    # ========================================================================

    @calibrated_method("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._embed_texts")
    def _embed_texts(self, texts: list[str]) -> NDArray[np.float32]:
        """Generate embeddings with caching and retry logic."""
        uncached_texts = []
        uncached_indices = []

        embeddings_list = []

        for i, text in enumerate(texts):
            text_hash = hashlib.sha256(text.encode()).hexdigest()[:16]

            if text_hash in self._embedding_cache:
                embeddings_list.append(self._embedding_cache[text_hash])
            else:
```

```
                uncached_texts.append(text)
                uncached_indices.append((i, text_hash))
                embeddings_list.append(None)  # Placeholder

        # Generate embeddings for uncached texts with retry logic
        if uncached_texts:
            if self.retry_handler:
                try:
                    from retry_handler import DependencyType

                    @self.retry_handler.with_retry(
                        DependencyType.EMBEDDING_SERVICE,
                        operation_name="encode_texts",
                        exceptions=(ConnectionError, TimeoutError, RuntimeError, OSError)
                    )
                    def encode_with_retry():
                        return self.embedding_model.encode(
                            uncached_texts,
                            batch_size=self.config.batch_size,
                            normalize_embeddings=self.config.normalize_embeddings,
                            show_progress_bar=False,
                            convert_to_numpy=True,
                        )

                    new_embeddings = encode_with_retry()
                except Exception as e:
                    self._logger.error(f"Failed to encode texts with retry: {e}")
                    raise
            else:
                new_embeddings = self.embedding_model.encode(
                    uncached_texts,
                    batch_size=self.config.batch_size,
                    normalize_embeddings=self.config.normalize_embeddings,
                    show_progress_bar=False,
                    convert_to_numpy=True,
                )

            # Cache and insert
            for (orig_idx, text_hash), emb in zip(uncached_indices, new_embeddings, strict=False):
                self._embedding_cache[text_hash] = emb
                embeddings_list[orig_idx] = emb

        return np.vstack(embeddings_list).astype(np.float32)

    def _filter_by_pdq(
        self, chunks: list[SemanticChunk], pdq_filter: PDQIdentifier
    ) -> list[SemanticChunk]:
        """Filter chunks by P-D-Q context."""

        def _repr_contract(value: Any) -> str:
            if value is None or isinstance(value, (int, float, bool)):
                return repr(value)
            if isinstance(value, str):
                # Strip excessive whitespace for logging clarity
                preview = value if len(value) <= 24 else f"{value[:21]}..."
                return repr(preview)
```

```
4970:             return type(value).__name__
4971:
4972:         def _log_mismatch(key: str, needed: Any, got: Any, index: int | None = None) -> None:
4973:             message = (
4974:                 "ERR_CONTRACT_MISMATCH[fn=_filter_by_pdq, "
4975:                 f"key='{key}', needed={_repr_contract(needed)}, got={_repr_contract(got)}"
4976:             )
4977:             if index is not None:
4978:                 message += f", index={index}"
4979:             message += "]"
4980:             self._logger.error(message)
4981:
4982:         self._logger.debug(
4983:             "edge %s â\206\222 _filter_by_pdq | params=%s",
4984:             self.__class__.__name__,
4985:             {
4986:                 "chunks_type": type(chunks).__name__,
4987:                 "chunks_len": len(chunks) if isinstance(chunks, list) else "n/a",
4988:                 "pdq_filter_type": type(pdq_filter).__name__,
4989:                 "pdq_filter_keys": sorted(pdq_filter.keys())
4990:                 if isinstance(pdq_filter, dict)
4991:                 else None,
4992:             },
4993:         )
4994:
4995:         if not isinstance(chunks, list):
4996:             _log_mismatch("chunks", "list", chunks)
4997:             return []
4998:
4999:         if not isinstance(pdq_filter, dict):
5000:             _log_mismatch("pdq_filter", "dict", pdq_filter)
5001:             return []
5002:
5003:         expected_policy = pdq_filter.get("policy")
5004:         expected_dimension = pdq_filter.get("dimension")
5005:
5006:         if expected_policy is None or expected_dimension is None:
5007:             _log_mismatch(
5008:                 "pdq_filter",
5009:                 "keys=('policy','dimension')",
5010:                 {"policy": expected_policy, "dimension": expected_dimension},
5011:             )
5012:             return []
5013:
5014:         filtered_chunks: list[SemanticChunk] = []
5015:
5016:         for index, chunk in enumerate(chunks):
5017:             if not isinstance(chunk, dict):
5018:                 _log_mismatch("chunk", "dict", chunk, index)
5019:                 continue
5020:
5021:             pdq_context = chunk.get("pdq_context")
5022:
5023:             if not pdq_context:
5024:                 _log_mismatch("pdq_context", True, pdq_context, index)
5025:                 continue
```

```
5026:
5027:             if not isinstance(pdq_context, dict):
5028:                 _log_mismatch("pdq_context", "dict", pdq_context, index)
5029:                 continue
5030:
5031:             policy = pdq_context.get("policy")
5032:             dimension = pdq_context.get("dimension")
5033:
5034:             if policy is None or dimension is None:
5035:                 _log_mismatch(
5036:                     "pdq_context",
5037:                     "keys=('policy','dimension')",
5038:                     {"policy": policy, "dimension": dimension},
5039:                     index,
5040:                 )
5041:                 continue
5042:
5043:             if policy == expected_policy and dimension == expected_dimension:
5044:                 filtered_chunks.append(chunk)
5045:
5046:         return filtered_chunks
5047:
5048:     def _apply_mmr(
5049:         self,
5050:         ranked_results: list[tuple[SemanticChunk, float]],
5051:     ) -> list[tuple[SemanticChunk, float]]:
5052:         """
5053:         Apply Maximal Marginal Relevance for diversification.
5054:
5055:         Balances relevance with diversity to avoid redundant results.
5056:         """
5057:         if len(ranked_results) <= 1:
5058:             return ranked_results
5059:
5060:         chunks, scores = zip(*ranked_results, strict=False)
5061:         chunk_embeddings = np.vstack([c["embedding"] for c in chunks])
5062:
5063:         selected_indices = []
5064:         remaining_indices = list(range(len(chunks)))
5065:
5066:         # Select first (most relevant)
5067:         selected_indices.append(0)
5068:         remaining_indices.remove(0)
5069:
5070:         # Iteratively select diverse documents
5071:         while remaining_indices and len(selected_indices) < len(chunks):
5072:             best_mmr_score = float("-inf")
5073:             best_idx = None
5074:
5075:             for idx in remaining_indices:
5076:                 # Relevance score
5077:                 relevance = scores[idx]
5078:
5079:                 # Diversity: max similarity to selected
5080:                 similarities_to_selected = cosine_similarity(
5081:                     chunk_embeddings[idx : idx + 1],
```

```
                    chunk_embeddings[selected_indices],
                ).max()

                # MMR score
                mmr_score = (
                    self.config.mmr_lambda * relevance
                    - (1 - self.config.mmr_lambda) * similarities_to_selected
                )

                if mmr_score > best_mmr_score:
                    best_mmr_score = mmr_score
                    best_idx = idx

            if best_idx is not None:
                selected_indices.append(best_idx)
                remaining_indices.remove(best_idx)

        # Reorder by MMR selection
        return [(chunks[i], scores[i]) for i in selected_indices]

    @calibrated_method("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._extract_numerical_values")
    def _extract_numerical_values(self, chunks: list[SemanticChunk]) -> list[float]:
        """
        Extract numerical values from chunks using advanced patterns.

        Focuses on policy-relevant metrics: percentages, amounts, counts.
        """
        numerical_values = []

        # Advanced patterns for Colombian policy metrics
        patterns = [
            r"(\d+(?:[.,]\d+)?)\s*%",  # Percentages
            r"\$\s*(\d{1,3}(?:[.,]\d{3})*(?:[.,]\d{2})?)",  # Currency
            # Millions
            r"(\d{1,3}(?:[.,]\d{3})*)\s*(?:millones?|mil\s+millones?)",
            # People count
            r"(\d+(?:[.,]\d+)?)\s*(?:personas|beneficiarios|habitantes)",
        ]

        for chunk in chunks:
            content = chunk["content"]

            for pattern in patterns:
                matches = re.finditer(pattern, content, re.IGNORECASE)

                for match in matches:
                    try:
                        # Extract and clean numerical string
                        raw_num = match.group(1)

                        # Handle Colombian and international decimal formats
                        if "." in raw_num and "," in raw_num:
                            # Colombian format: dot as thousands, comma as decimal
                            num_str = raw_num.replace(".", "").replace(",", ".")
                        elif "," in raw_num:
                            # Comma as decimal separator
```

```
                            num_str = raw_num.replace(",", ".")
                        else:
                            # Only dot or plain number
                            num_str = raw_num

                        value = float(num_str)

                        # Normalize to 0-1 scale if it's a percentage
                        if "%" in match.group(0) and value <= 100:
                            value = value / 100.0

                        # Filter outliers
                        if 0 <= value <= 1e9:  # Reasonable range
                            numerical_values.append(value)

                    except (ValueError, IndexError):
                        continue

        return numerical_values

    @calibrated_method("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq")
    def _generate_query_from_pdq(self, pdq: PDQIdentifier) -> str:
        """Generate search query from P-D-Q identifier."""
        policy_name = PolicyDomain[pdq["policy"]].value
        dimension_name = AnalyticalDimension[pdq["dimension"]].value

        query = f"{policy_name} - {dimension_name}"
        return query

    def _compute_overall_confidence(
        self,
        relevant_chunks: list[tuple[SemanticChunk, float]],
        numerical_eval: BayesianEvaluation,
    ) -> float:
        """
        Compute overall confidence score combining semantic and numerical evidence.

        Considers:
        - Number of relevant chunks
        - Semantic relevance scores
        - Numerical evidence strength
        - Statistical coherence
        """
        if not relevant_chunks:
            return ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1509_19", 0
.0)

        # Semantic confidence: average of top scores
        semantic_scores = [score for _, score in relevant_chunks[:5]]
        semantic_confidence = (
            float(np.mean(semantic_scores)) if semantic_scores else ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._g
enerate_query_from_pdq", "auto_param_L1514_68", 0.0)
        )

        # Numerical confidence: based on evidence strength and coherence
        evidence_strength_map = {
```

```
            "weak": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1519_20", 0.25),
            "moderate": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1520_24", 0.5),
            "strong": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1521_22", 0.75),
            "very_strong": ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1522_27", 1.0),
        }
        numerical_confidence = (
            evidence_strength_map[numerical_eval["evidence_strength"]]
            * numerical_eval["numerical_coherence"]
        )

        # Combined confidence: weighted average
        overall_confidence = ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1530_29", 0.6) * semantic_confidence + ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1530_57", 0.4) * numerical_confidence

        return float(np.clip(overall_confidence, ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1532_49", 0.0), ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq", "auto_param_L1532_54", 1.0)))

    @lru_cache(maxsize=1024)
    @calibrated_method("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder._cached_similarity")
    def _cached_similarity(self, text_hash1: str, text_hash2: str) -> float:
        """Cached similarity computation for performance.
        Assumes embeddings are cached in self._embedding_cache using text_hash as key.
        """
        emb1 = self._embedding_cache[text_hash1]
        emb2 = self._embedding_cache[text_hash2]
        return float(cosine_similarity(emb1.reshape(1, -1), emb2.reshape(1, -1))[0, 0])

    @calibrated_method("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder.get_diagnostics")
    def get_diagnostics(self) -> dict[str, Any]:
        """Get system diagnostics and performance metrics."""
        return {
            "model": self.config.embedding_model,
            "embedding_cache_size": len(self._embedding_cache),
            "chunk_cache_size": len(self._chunk_cache),
            "total_chunks_processed": sum(
                len(chunks) for chunks in self._chunk_cache.values()
            ),
            "config": {
                "chunk_size": self.config.chunk_size,
                "chunk_overlap": self.config.chunk_overlap,
                "top_k_candidates": self.config.top_k_candidates,
                "top_k_rerank": self.config.top_k_rerank,
                "mmr_lambda": self.config.mmr_lambda,
            },
        }

# ============================================================================
# PRODUCTION FACTORY AND UTILITIES
# ============================================================================

```

```
5240: def create_policy_embedder(
5241:     model_tier: Literal["fast", "balanced", "accurate"] = "balanced",
5242: ) -> PolicyAnalysisEmbedder:
5243:     """
5244:     Factory function for creating production-ready policy embedder.
5245:
5246:     Args:
5247:         model_tier: Performance/accuracy trade-off
5248:             - "fast": Lightweight, low latency
5249:             - "balanced": Good performance/accuracy balance (default)
5250:             - "accurate": Maximum accuracy, higher latency
5251:
5252:     Returns:
5253:         Configured PolicyAnalysisEmbedder instance
5254:     """
5255:     model_configs = {
5256:         "fast": PolicyEmbeddingConfig(
5257:             embedding_model="sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2",
5258:             cross_encoder_model=DEFAULT_CROSS_ENCODER_MODEL,
5259:             chunk_size=256,
5260:             chunk_overlap=64,
5261:             top_k_candidates=30,
5262:             top_k_rerank=5,
5263:             batch_size=64,
5264:         ),
5265:         "balanced": PolicyEmbeddingConfig(
5266:             embedding_model=MODEL_PARAPHRASE_MULTILINGUAL,
5267:             cross_encoder_model=DEFAULT_CROSS_ENCODER_MODEL,
5268:             chunk_size=512,
5269:             chunk_overlap=128,
5270:             top_k_candidates=50,
5271:             top_k_rerank=10,
5272:             batch_size=32,
5273:         ),
5274:         "accurate": PolicyEmbeddingConfig(
5275:             embedding_model=MODEL_PARAPHRASE_MULTILINGUAL,
5276:             cross_encoder_model="cross-encoder/mmarco-mMiniLMv2-L12-H384-v1",
5277:             chunk_size=768,
5278:             chunk_overlap=192,
5279:             top_k_candidates=100,
5280:             top_k_rerank=20,
5281:             batch_size=16,
5282:         ),
5283:     }
5284:
5285:     config = model_configs[model_tier]
5286:
5287:     logger = logging.getLogger("PolicyEmbedderFactory")
5288:     logger.info("Creating policy embedder with tier: %s", model_tier)
5289:
5290:     return PolicyAnalysisEmbedder(config)
5291:
5292: # ============================================================================
5293: # PRODUCER CLASS - Registry Exposure
5294: # ============================================================================
5295:
```

```
5296: class EmbeddingPolicyProducer:
5297:     """
5298:     Producer wrapper for embedding policy analysis with registry exposure
5299:
5300:     Provides public API methods for orchestrator integration without exposing
5301:     internal implementation details or summarization logic.
5302:
5303:     Version: ParameterLoaderV2.get("farfan_core.processing.embedding_policy.PolicyAnalysisEmbedder.get_diagnostics", "auto_param_L1630_13", 1.0).0
5304:     Producer Type: Embedding / Semantic Search
5305:     """
5306:
5307:     def __init__(
5308:         self,
5309:         config: PolicyEmbeddingConfig | None = None,
5310:         model_tier: Literal["fast", "balanced", "accurate"] = "balanced",
5311:         retry_handler=None
5312:     ) -> None:
5313:         """Initialize producer with optional configuration"""
5314:         if config is None:
5315:             self.embedder = create_policy_embedder(model_tier)
5316:         else:
5317:             self.embedder = PolicyAnalysisEmbedder(config, retry_handler=retry_handler)
5318:
5319:         self._logger = logging.getLogger(self.__class__.__name__)
5320:         self._logger.info("EmbeddingPolicyProducer initialized")
5321:
5322:     # ========================================================================
5323:     # DOCUMENT PROCESSING API
5324:     # ========================================================================
5325:
5326:     def process_document(
5327:         self,
5328:         document_text: str,
5329:         document_metadata: dict[str, Any]
5330:     ) -> list[SemanticChunk]:
5331:         """Process document into semantic chunks with embeddings"""
5332:         return self.embedder.process_document(document_text, document_metadata)
5333:
5334:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_chunk_count")
5335:     def get_chunk_count(self, chunks: list[SemanticChunk]) -> int:
5336:         """Get number of chunks"""
5337:         return len(chunks)
5338:
5339:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_chunk_text")
5340:     def get_chunk_text(self, chunk: SemanticChunk) -> str:
5341:         """Extract text from chunk"""
5342:         return chunk["content"]
5343:
5344:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_chunk_embedding")
5345:     def get_chunk_embedding(self, chunk: SemanticChunk) -> NDArray[np.float32]:
5346:         """Extract embedding from chunk"""
5347:         return chunk["embedding"]
5348:
5349:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_chunk_metadata")
5350:     def get_chunk_metadata(self, chunk: SemanticChunk) -> dict[str, Any]:
5351:         """Extract metadata from chunk"""
```

```
5352:         return chunk["metadata"]
5353:
5354:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_chunk_pdq_context")
5355:     def get_chunk_pdq_context(self, chunk: SemanticChunk) -> PDQIdentifier | None:
5356:         """Extract P-D-Q context from chunk"""
5357:         return chunk["pdq_context"]
5358:
5359:     # ========================================================================
5360:     # SEMANTIC SEARCH API
5361:     # ========================================================================
5362:
5363:     def semantic_search(
5364:         self,
5365:         query: str,
5366:         document_chunks: list[SemanticChunk],
5367:         pdq_filter: PDQIdentifier | None = None,
5368:         use_reranking: bool = True
5369:     ) -> list[tuple[SemanticChunk, float]]:
5370:         """Advanced semantic search with reranking"""
5371:         return self.embedder.semantic_search(
5372:             query, document_chunks, pdq_filter, use_reranking
5373:         )
5374:
5375:     def get_search_result_chunk(
5376:         self, result: tuple[SemanticChunk, float]
5377:     ) -> SemanticChunk:
5378:         """Extract chunk from search result"""
5379:         return result[0]
5380:
5381:     def get_search_result_score(
5382:         self, result: tuple[SemanticChunk, float]
5383:     ) -> float:
5384:         """Extract relevance score from search result"""
5385:         return result[1]
5386:
5387:     # ========================================================================
5388:     # P-D-Q ANALYSIS API
5389:     # ========================================================================
5390:
5391:     def generate_pdq_report(
5392:         self,
5393:         document_chunks: list[SemanticChunk],
5394:         target_pdq: PDQIdentifier
5395:     ) -> dict[str, Any]:
5396:         """Generate comprehensive analytical report for P-D-Q question"""
5397:         return self.embedder.generate_pdq_report(document_chunks, target_pdq)
5398:
5399:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_pdq_evidence_count")
5400:     def get_pdq_evidence_count(self, report: dict[str, Any]) -> int:
5401:         """Extract evidence count from P-D-Q report"""
5402:         return report.get("evidence_count", 0)
5403:
5404:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_pdq_numerical_evaluation")
5405:     def get_pdq_numerical_evaluation(self, report: dict[str, Any]) -> dict[str, Any]:
5406:         """Extract numerical evaluation from P-D-Q report"""
5407:         return report.get("numerical_evaluation", {})
```

```
5408:
5409:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_pdq_evidence_passages")
5410:     def get_pdq_evidence_passages(self, report: dict[str, Any]) -> list[dict[str, Any]]:
5411:         """Extract evidence passages from P-D-Q report"""
5412:         return report.get("evidence_passages", [])
5413:
5414:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_pdq_confidence")
5415:     def get_pdq_confidence(self, report: dict[str, Any]) -> float:
5416:         """Extract confidence from P-D-Q report"""
5417:         return report.get("confidence", ParameterLoaderV2.get("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_pdq_confidence", "auto_pa
ram_L1744_40", 0.0))
5418:
5419:     # ========================================================================
5420:     # BAYESIAN NUMERICAL ANALYSIS API
5421:     # ========================================================================
5422:
5423:     def evaluate_numerical_consistency(
5424:         self,
5425:         chunks: list[SemanticChunk],
5426:         pdq_context: PDQIdentifier
5427:     ) -> BayesianEvaluation:
5428:         """Evaluate numerical consistency with Bayesian analysis"""
5429:         return self.embedder.evaluate_policy_numerical_consistency(
5430:             chunks, pdq_context
5431:         )
5432:
5433:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_point_estimate")
5434:     def get_point_estimate(self, evaluation: BayesianEvaluation) -> float:
5435:         """Extract point estimate from Bayesian evaluation"""
5436:         return evaluation["point_estimate"]
5437:
5438:     def get_credible_interval(
5439:         self, evaluation: BayesianEvaluation
5440:     ) -> tuple[float, float]:
5441:         """Extract 95% credible interval from Bayesian evaluation"""
5442:         return evaluation["credible_interval_95"]
5443:
5444:     def get_evidence_strength(
5445:         self, evaluation: BayesianEvaluation
5446:     ) -> Literal["weak", "moderate", "strong", "very_strong"]:
5447:         """Extract evidence strength classification"""
5448:         return evaluation["evidence_strength"]
5449:
5450:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_numerical_coherence")
5451:     def get_numerical_coherence(self, evaluation: BayesianEvaluation) -> float:
5452:         """Extract numerical coherence score"""
5453:         return evaluation["numerical_coherence"]
5454:
5455:     # ========================================================================
5456:     # POLICY COMPARISON API
5457:     # ========================================================================
5458:
5459:     def compare_policy_interventions(
5460:         self,
5461:         intervention_a_chunks: list[SemanticChunk],
5462:         intervention_b_chunks: list[SemanticChunk],
```

```
5463:         pdq_context: PDQIdentifier
5464:     ) -> dict[str, Any]:
5465:         """Bayesian comparison of two policy interventions"""
5466:         return self.embedder.compare_policy_interventions(
5467:             intervention_a_chunks, intervention_b_chunks, pdq_context
5468:         )
5469:
5470:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_probability")
5471:     def get_comparison_probability(self, comparison: dict[str, Any]) -> float:
5472:         """Extract probability that A is better than B"""
5473:         return comparison.get("probability_a_better", ParameterLoaderV2.get("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_
probability", "auto_param_L1800_54", 0.5))
5474:
5475:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_bayes_factor")
5476:     def get_comparison_bayes_factor(self, comparison: dict[str, Any]) -> float:
5477:         """Extract Bayes factor from comparison"""
5478:         return comparison.get("bayes_factor", ParameterLoaderV2.get("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_bayes_fa
ctor", "auto_param_L1805_46", 1.0))
5479:
5480:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_difference_mean")
5481:     def get_comparison_difference_mean(self, comparison: dict[str, Any]) -> float:
5482:         """Extract mean difference from comparison"""
5483:         return comparison.get("difference_mean", ParameterLoaderV2.get("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_diffe
rence_mean", "auto_param_L1810_49", 0.0))
5484:
5485:     # ========================================================================
5486:     # UTILITY API
5487:     # ========================================================================
5488:
5489:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_diagnostics")
5490:     def get_diagnostics(self) -> dict[str, Any]:
5491:         """Get system diagnostics and performance metrics"""
5492:         return self.embedder.get_diagnostics()
5493:
5494:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_config")
5495:     def get_config(self) -> PolicyEmbeddingConfig:
5496:         """Get current configuration"""
5497:         return self.embedder.config
5498:
5499:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.list_policy_domains")
5500:     def list_policy_domains(self) -> list[PolicyDomain]:
5501:         """List all policy domains"""
5502:         return list(PolicyDomain)
5503:
5504:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.list_analytical_dimensions")
5505:     def list_analytical_dimensions(self) -> list[AnalyticalDimension]:
5506:         """List all analytical dimensions"""
5507:         return list(AnalyticalDimension)
5508:
5509:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_policy_domain_description")
5510:     def get_policy_domain_description(self, domain: PolicyDomain) -> str:
5511:         """Get description for policy domain"""
5512:         return domain.value
5513:
5514:     @calibrated_method("farfan_core.processing.embedding_policy.EmbeddingPolicyProducer.get_analytical_dimension_description")
5515:     def get_analytical_dimension_description(self, dimension: AnalyticalDimension) -> str:
```

```
        """Get description for analytical dimension"""
        return dimension.value

    def create_pdq_identifier(
        self,
        policy: str,
        dimension: str,
        question: int
    ) -> PDQIdentifier:
        """Create P-D-Q identifier"""
        return PDQIdentifier(
            question_unique_id=f"{policy}-{dimension}-Q{question}",
            policy=policy,
            dimension=dimension,
            question=question,
            rubric_key=f"{dimension}-Q{question}"
        )

# ============================================================================
# COMPREHENSIVE EXAMPLE - Production Usage
# ============================================================================

def example_pdm_analysis() -> None:
    """
    Complete example: analyzing Colombian Municipal Development Plan.
    """
    import logging

    logging.basicConfig(level=logging.INFO)

    # Sample PDM excerpt (simplified)
    pdm_document = """
    PLAN DE DESARROLLO MUNICIPAL 2024-2027
    MUNICIPIO DE EJEMPLO, COLOMBIA

    EJE ESTRATÃ\211GICO 1: DERECHOS DE LAS MUJERES E IGUALDAD DE GÃ\211NERO

    DIAGNÃ\223STICO
    El municipio presenta una brecha de gÃ©nero del 18.5% en participaciÃ³n laboral.
    Se identificaron 2,340 mujeres en situaciÃ³n de vulnerabilidad econÃ³mica.
    El presupuesto asignado asciende a $450 millones para el cuatrienio.

    DISEÃ\221O DE INTERVENCIÃ\223N
    Se implementarÃ¡n 3 programas de empoderamiento econÃ³mico:
    - Programa de formaciÃ³n tÃ©cnica: 500 beneficiarias
    - MicrocrÃ©ditos productivos: $280 millones
    - Fortalecimiento empresarial: 150 emprendimientos

    PRODUCTOS Y OUTPUTS
    Meta cuatrienio: reducir brecha de gÃ©nero al 12% (reducciÃ³n del 35.1%)
    Indicador: Tasa de participaciÃ³n laboral femenina
    LÃ-nea base: 42.3% | Meta: 55.8%

    RESULTADOS ESPERADOS
    Incremento del 25% en ingresos promedio de beneficiarias
    CreaciÃ³n de 320 nuevos empleos formales para mujeres
```

```
    Sostenibilidad: 78% de emprendimientos activos a 2 aÃ±os
    """

    metadata = {
        "doc_id": "PDM_EJEMPLO_2024_2027",
        "municipality": "Ejemplo",
        "department": "Ejemplo",
        "year": 2024,
    }

    # Create embedder
    print("=" * 80)
    print("POLICY ANALYSIS EMBEDDER - PRODUCTION EXAMPLE")
    print("=" * 80)

    embedder = create_policy_embedder(model_tier="balanced")

    # Process document
    print("\n1. PROCESSING DOCUMENT")
    chunks = embedder.process_document(pdm_document, metadata)
    print(f"   Generated {len(chunks)} semantic chunks")

    # Define P-D-Q query
    pdq_query = PDQIdentifier(
        question_unique_id="P1-D1-Q3",
        policy="P1",
        dimension="D1",
        question=3,
        rubric_key="D1-Q3",
    )

    print(f"\n2. ANALYZING P-D-Q: {pdq_query['question_unique_id']}")
    print(f"   Policy: {PolicyDomain.P1.value}")
    print(f"   Dimension: {AnalyticalDimension.D1.value}")

    # Generate comprehensive report
    report = embedder.generate_pdq_report(chunks, pdq_query)

    print("\n3. ANALYSIS RESULTS")
    print(f"   Evidence chunks found: {report['evidence_count']}")
    print(f"   Overall confidence: {report['confidence']:.3f}")
    print("\n   Numerical Evaluation:")
    print(
        f"   - Point estimate: {report['numerical_evaluation']['point_estimate']:.3f}"
    )
    print(
        f"   - 95% CI: [{report['numerical_evaluation']['credible_interval_95'][0]:.3f}, "
        f"{report['numerical_evaluation']['credible_interval_95'][1]:.3f}]"
    )
    print(
        f"   - Evidence strength: {report['numerical_evaluation']['evidence_strength']}"
    )
    print(
        f"   - Numerical coherence: {report['numerical_evaluation']['numerical_coherence']:.3f}"
    )

```

```
    print("\n4. TOP EVIDENCE PASSAGES:")
    for i, passage in enumerate(report["evidence_passages"], 1):
        print(f"\n   [{i}] Relevance: {passage['relevance_score']:.3f}")
        print(f"       {passage['content'][:200]}...")

    # System diagnostics
    print("\n5. SYSTEM DIAGNOSTICS")
    diag = embedder.get_diagnostics()
    print(f"   Model: {diag['model']}")
    print(f"   Cache efficiency: {diag['embedding_cache_size']} embeddings cached")
    print(f"   Total chunks processed: {diag['total_chunks_processed']}")

    print("\n" + "=" * 80)
    print("ANALYSIS COMPLETE")
    print("=" * 80)



================================================================================
FILE: src/farfan_pipeline/processing/models.py
================================================================================

"""
CPP Ingestion Models (Deprecated - Use SPC)

Data models for Canon Policy Package (CPP) ingestion pipeline.
These models define the structure of policy documents after phase-one ingestion.

NOTE: This is a compatibility layer. New code should use SPC (Smart Policy Chunks) terminology.
"""

from __future__ import annotations

from dataclasses import dataclass, field
from enum import Enum
from typing import Any
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method


class ChunkResolution(Enum):
    """Granularity level for policy chunks."""
    MICRO = "MICRO"  # Fine-grained chunks (sentences, clauses)
    MESO = "MESO"    # Medium chunks (paragraphs, sections)
    MACRO = "MACRO"  # Coarse chunks (chapters, themes)


@dataclass
class TextSpan:
    """Represents a span of text in the original document."""
    start: int
    end: int


@dataclass
class Confidence:
```

```
5684:     """Confidence scores for various extraction processes."""
5685:     layout: float = 1.0
5686:     ocr: float = 1.0
5687:     typing: float = 1.0
5688:
5689:
5690: @dataclass
5691: class PolicyFacet:
5692:     """Policy-related metadata facets."""
5693:     programs: list[str] = field(default_factory=list)
5694:     projects: list[str] = field(default_factory=list)
5695:     axes: list[str] = field(default_factory=list)
5696:
5697:
5698: @dataclass
5699: class TimeFacet:
5700:     """Temporal metadata facets."""
5701:     years: list[int] = field(default_factory=list)
5702:     periods: list[str] = field(default_factory=list)
5703:
5704:
5705: @dataclass
5706: class GeoFacet:
5707:     """Geographic metadata facets."""
5708:     territories: list[str] = field(default_factory=list)
5709:     regions: list[str] = field(default_factory=list)
5710:
5711:
5712: @dataclass
5713: class ProvenanceMap:
5714:     """Provenance information for chunk extraction."""
5715:     source_page: int | None = None
5716:     source_section: str | None = None
5717:     extraction_method: str = "semantic_chunking"
5718:
5719:
5720: @dataclass
5721: class Budget:
5722:     """Budget information extracted from policy document."""
5723:     source: str
5724:     use: str
5725:     amount: float
5726:     year: int
5727:     currency: str = "COP"
5728:
5729:
5730: # Alias for compatibility
5731: BudgetInfo = Budget
5732:
5733:
5734: @dataclass
5735: class KPI:
5736:     """Key Performance Indicator extracted from policy."""
5737:     indicator_name: str
5738:     target_value: float | None = None
5739:     unit: str | None = None
```

```
    year: int | None = None


@dataclass
class Entity:
    """Named entity extracted from text."""
    text: str
    entity_type: str
    confidence: float = 1.0


@dataclass
class Chunk:
    """
    A semantic chunk of policy text with metadata.

    This is the fundamental unit of the CPP/SPC ingestion pipeline.
    """
    id: str
    text: str
    text_span: TextSpan
    resolution: ChunkResolution
    bytes_hash: str
    policy_area_id: str | None = None  # PA01-PA10 canonical code
    dimension_id: str | None = None    # DIM01-DIM06 canonical code

    # Facets and metadata
    policy_facets: PolicyFacet = field(default_factory=PolicyFacet)
    time_facets: TimeFacet = field(default_factory=TimeFacet)
    geo_facets: GeoFacet = field(default_factory=GeoFacet)
    confidence: Confidence = field(default_factory=Confidence)

    # Optional structured data
    provenance: ProvenanceMap | None = None
    budget: Budget | None = None
    kpi: KPI | None = None
    entities: list[Entity] = field(default_factory=list)


@dataclass
class ChunkGraph:
    """
    Graph structure containing all chunks and their relationships.
    """
    chunks: dict[str, Chunk] = field(default_factory=dict)
    edges: list[tuple[str, str, str]] = field(default_factory=list)  # (from_id, to_id, relation_type)

    @calibrated_method("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_chunk")
    def add_chunk(self, chunk: Chunk) -> None:
        """Add a chunk to the graph."""
        self.chunks[chunk.id] = chunk

    @calibrated_method("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge")
    def add_edge(self, from_id: str, to_id: str, relation_type: str) -> None:
        """Add an edge to the graph."""
        self.edges.append((from_id, to_id, relation_type))
```

```
5796:
5797:
5798: @dataclass
5799: class PolicyManifest:
5800:     """
5801:     High-level manifest summarizing policy structure.
5802:     """
5803:     axes: list[str] = field(default_factory=list)
5804:     programs: list[str] = field(default_factory=list)
5805:     projects: list[str] = field(default_factory=list)
5806:     years: list[int] = field(default_factory=list)
5807:     territories: list[str] = field(default_factory=list)
5808:     indicators: list[str] = field(default_factory=list)
5809:     budget_rows: int = 0
5810:
5811:
5812: @dataclass
5813: class QualityMetrics:
5814:     """
5815:     Quality metrics for the ingestion process.
5816:     """
5817:     boundary_f1: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L167_25", 0.0)
5818:     kpi_linkage_rate: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L168_30", 0.0)
5819:     budget_consistency_score: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L169_38", 0.0)
5820:     provenance_completeness: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L170_37", 0.0)
5821:     structural_consistency: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L171_36", 0.0)
5822:     temporal_robustness: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L172_33", 0.0)
5823:     chunk_context_coverage: float = ParameterLoaderV2.get("farfan_core.processing.cpp_ingestion.models.ChunkGraph.add_edge", "auto_param_L173_36", 0.0)
5824:
5825:
5826: @dataclass
5827: class IntegrityIndex:
5828:     """
5829:     Cryptographic integrity verification data.
5830:
5831:     Uses BLAKE2b (not BLAKE3) for aggregate hash computation.
5832:     Implementation uses hashlib.blake2b over JSON-serialized chunk hashes.
5833:     """
5834:     blake2b_root: str  # Aggregate hash (BLAKE2b-256) of all chunk hashes
5835:     chunk_hashes: dict[str, str] = field(default_factory=dict)
5836:
5837:
5838: @dataclass
5839: class CanonPolicyPackage:
5840:     """
5841:     Canon Policy Package - Complete output from phase-one ingestion.
5842:
5843:     This is the top-level container for all ingestion results.
5844:     Also known as Smart Policy Chunks (SPC) in newer terminology.
5845:     """
5846:     schema_version: str
5847:     chunk_graph: ChunkGraph
5848:
5849:     # Optional high-level metadata
5850:     policy_manifest: PolicyManifest | None = None
5851:     quality_metrics: QualityMetrics | None = None
```

```
    integrity_index: IntegrityIndex | None = None

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


================================================================================
FILE: src/farfan_pipeline/processing/policy_processor.py
================================================================================

"""
Causal Framework Policy Plan Processor - Industrial Grade
==========================================================

A mathematically rigorous, production-hardened system for extracting and
validating causal evidence from Colombian local development plans against
the DECALOGO framework's six-dimensional evaluation criteria.

Architecture:
    - Bayesian evidence accumulation for probabilistic confidence scoring
    - Multi-scale text segmentation with coherence-preserving boundaries
    - Differential privacy-aware pattern matching for reproducibility
    - Entropy-based relevance ranking with TF-IDF normalization
    - Graph-theoretic dependency validation for causal chain integrity

Version: 3.0.0 | ISO 9001:2015 Compliant
Author: Policy Analytics Research Unit
License: Proprietary
"""

import logging
import re
import unicodedata
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from enum import Enum
from functools import lru_cache
from pathlib import Path
from typing import Any, ClassVar, Optional

import numpy as np

# Import runtime error fixes for defensive programming
from farfan_pipeline.utils.runtime_error_fixes import ensure_list_return

from farfan_pipeline.analysis.financiero_viabilidad_tablas import PDETAnalysisException, QualityScore
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method
from farfan_pipeline.core.ports import (
    PortDocumentLoader,
    PortMunicipalOntology,
    PortSemanticAnalyzer,
    PortPerformanceAnalyzer,
    PortContradictionDetector,
    PortTemporalLogicVerifier,
```

```
    PortBayesianConfidenceCalculator,
    PortMunicipalAnalyzer,
)

try:
    from farfan_pipeline.analysis.contradiction_deteccion import (
        PolicyDimension as ContradictionPolicyDimension,
    )
    CONTRADICTION_MODULE_AVAILABLE = True
except Exception as import_error:
    CONTRADICTION_MODULE_AVAILABLE = False
    logger = logging.getLogger(__name__)
    logger.warning(
        "Falling back to lightweight contradiction components due to import error: %s",
        import_error,
    )

    class ContradictionPolicyDimension(Enum):  # type: ignore[misc]
        DIAGNOSTICO = "diagnÃ³stico"
        ESTRATEGICO = "estratÃ©gico"
        PROGRAMATICO = "programÃ¡tico"
        FINANCIERO = "plan plurianual de inversiones"
        SEGUIMIENTO = "seguimiento y evaluaciÃ³n"
        TERRITORIAL = "ordenamiento territorial"


class _FallbackBayesianCalculator:
    """Fallback Bayesian calculator when advanced module is unavailable."""

    def __init__(self) -> None:
        self.prior_alpha = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackBayesianCalculator.__init__", "auto_param_L64_31", 1.0)
        self.prior_beta = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackBayesianCalculator.__init__", "auto_param_L65_30", 1.0)

    def calculate_posterior(
        self, evidence_strength: float, observations: int, domain_weight: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackBa
yesianCalculator.__init__", "auto_param_L68_86", 1.0)
    ) -> float:
        alpha_post = self.prior_alpha + evidence_strength * observations * domain_weight
        beta_post = self.prior_beta + (1 - evidence_strength) * observations * domain_weight
        return alpha_post / (alpha_post + beta_post)


class _FallbackTemporalVerifier:
    """Fallback temporal verifier providing graceful degradation."""

    @calibrated_method("farfan_core.processing.policy_processor._FallbackTemporalVerifier.verify_temporal_consistency")
    def verify_temporal_consistency(self, statements: list[Any]) -> tuple[bool, list[dict[str, Any]]]:
        return True, []


class _FallbackContradictionDetector:
    """Fallback contradiction detector providing graceful degradation."""

    def detect(
        self,
        text: str,
```

```
5963:         plan_name: str = "PDM",
5964:         dimension: Any = None,
5965:     ) -> dict[str, Any]:
5966:         return {
5967:             "plan_name": plan_name,
5968:             "dimension": getattr(dimension, "value", "unknown"),
5969:             "contradictions": [],
5970:             "total_contradictions": 0,
5971:             "high_severity_count": 0,
5972:             "coherence_metrics": {},
5973:             "recommendations": [],
5974:             "knowledge_graph_stats": {"nodes": 0, "edges": 0, "components": 0},
5975:         }
5976:
5977:     @calibrated_method("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements")
5978:     def _extract_policy_statements(self, text: str, dimension: Any) -> list[Any]:
5979:         return []
5980:
5981: # ============================================================================
5982: # LOGGING CONFIGURATION
5983: # ============================================================================
5984: # Note: logging.basicConfig should be called by the application entry point,
5985: # not at module import time to avoid side effects
5986: logger = logging.getLogger(__name__)
5987:
5988: # ============================================================================
5989: # CAUSAL DIMENSION TAXONOMY (DECALOGO Framework)
5990: # ============================================================================
5991:
5992: class CausalDimension(Enum):
5993:     """Six-dimensional causal framework taxonomy aligned with DECALOGO."""
5994:
5995:     D1_INSUMOS = "d1_insumos"
5996:     D2_ACTIVIDADES = "d2_actividades"
5997:     D3_PRODUCTOS = "d3_productos"
5998:     D4_RESULTADOS = "d4_resultados"
5999:     D5_IMPACTOS = "d5_impactos"
6000:     D6_CAUSALIDAD = "d6_causalidad"
6001:
6002: # ============================================================================
6003: # ENHANCED PATTERN LIBRARY WITH SEMANTIC HIERARCHIES
6004: # ============================================================================
6005:
6006: CAUSAL_PATTERN_TAXONOMY: dict[CausalDimension, dict[str, list[str]]] = {
6007:     CausalDimension.D1_INSUMOS: {
6008:         "diagnostico_cuantitativo": [
6009:             r"\b(?:diagn[Ã³o]stico\s+(?:cuantitativo|estad[Ã-i]stico|situacional))\b",
6010:             r"\b(?:an[Ã¡a]lisis\s+(?:de\s+)?(?:brecha|situaci[Ã³o]n\s+actual))\b",
6011:             r"\b(?:caracterizaci[Ã³o]n\s+(?:territorial|poblacional|sectorial))\b",
6012:         ],
6013:         "lineas_base_temporales": [
6014:             r"\b(?:l[Ã-i]nea(?:s)?\s+(?:de\s+)?base)\b",
6015:             r"\b(?:valor(?:es)?\s+inicial(?:es)?)\b",
6016:             r"\b(?:serie(?:s)?\s+(?:hist[Ã³o]rica(?:s)?|temporal(?:es)?))\b",
6017:             r"\b(?:medici[Ã³o]n\s+(?:de\s+)?referencia)\b",
6018:         ],
```

```
        "recursos_programaticos": [
            r"\b(?:presupuesto\s+(?:plurianual|de\s+inversi[Ã³o]n))\b",
            r"\b(?:plan\s+(?:plurianual|financiero|operativo\s+anual))\b",
            r"\b(?:marco\s+fiscal\s+de\s+mediano\s+plazo)\b",
            r"\b(?:trazabilidad\s+(?:presupuestal|program[Ã¡a]tica))\b",
        ],
        "capacidad_institucional": [
            r"\b(?:capacidad(?:es)?\s+(?:institucional(?:es)?|t[Ã©e]cnica(?:s)?))\b",
            r"\b(?:talento\s+humano\s+(?:disponible|requerido))\b",
            r"\b(?:gobernanza\s+(?:de\s+)?(?:datos|informaci[Ã³o]n))\b",
            r"\b(?:brechas?\s+(?:de\s+)?implementaci[Ã³o]n)\b",
        ],
    },
    CausalDimension.D2_ACTIVIDADES: {
        "formalizacion_actividades": [
            r"\b(?:plan\s+de\s+acci[Ã³o]n\s+detallado)\b",
            r"\b(?:matriz\s+de\s+(?:actividades|intervenciones))\b",
            r"\b(?:cronograma\s+(?:de\s+)?ejecuci[Ã³o]n)\b",
            r"\b(?:responsables?\s+(?:designados?|identificados?))\b",
        ],
        "mecanismo_causal": [
            r"\b(?:mecanismo(?:s)?\s+causal(?:es)?)\b",
            r"\b(?:teor[Ã-i]a\s+(?:de\s+)?intervenci[Ã³o]n)\b",
            r"\b(?:cadena\s+(?:de\s+)?causaci[Ã³o]n)\b",
            r"\b(?:v[Ã-i]nculo(?:s)?\s+explicativo(?:s)?)\b",
        ],
        "poblacion_objetivo": [
            r"\b(?:poblaci[Ã³o]n\s+(?:diana|objetivo|beneficiaria))\b",
            r"\b(?:criterios?\s+de\s+focalizaci[Ã³o]n)\b",
            r"\b(?:segmentaci[Ã³o]n\s+(?:territorial|poblacional))\b",
        ],
        "dosificacion_intervencion": [
            r"\b(?:dosificaci[Ã³o]n\s+(?:de\s+)?(?:la\s+)?intervenci[Ã³o]n)\b",
            r"\b(?:intensidad\s+(?:de\s+)?tratamiento)\b",
            r"\b(?:duraci[Ã³o]n\s+(?:de\s+)?exposici[Ã³o]n)\b",
        ],
    },
    CausalDimension.D3_PRODUCTOS: {
        "indicadores_producto": [
            r"\b(?:indicador(?:es)?\s+de\s+(?:producto|output|gesti[Ã³o]n))\b",
            r"\b(?:entregables?\s+verificables?)\b",
            r"\b(?:metas?\s+(?:de\s+)?producto)\b",
        ],
        "verificabilidad": [
            r"\b(?:f[Ã³o]rmula\s+(?:de\s+)?(?:c[Ã¡a]lculo|medici[Ã³o]n))\b",
            r"\b(?:fuente(?:s)?\s+(?:de\s+)?verificaci[Ã³o]n)\b",
            r"\b(?:medio(?:s)?\s+de\s+(?:prueba|evidencia))\b",
        ],
        "trazabilidad_producto": [
            r"\b(?:trazabilidad\s+(?:de\s+)?productos?)\b",
            r"\b(?:sistema\s+de\s+registro)\b",
            r"\b(?:cobertura\s+(?:real|efectiva))\b",
        ],
    },
    CausalDimension.D4_RESULTADOS: {
        "metricas_outcome": [
```

```
            r"\b(?:(?:indicador(?:es)?|m[Ã©e]trica(?:s)?)\s+de\s+(?:resultado|outcome))\b",
            r"\b(?:criterios?\s+de\s+[Ã©e]xito)\b",
            r"\b(?:umbral(?:es)?\s+de\s+desempe[Ã±n]o)\b",
        ],
        "encadenamiento_causal": [
            r"\b(?:encadenamiento\s+(?:causal|l[Ã³o]gico))\b",
            r"\b(?:ruta(?:s)?\s+cr[Ãi]tica(?:s)?)\b",
            r"\b(?:dependencias?\s+causales?)\b",
        ],
        "ventana_maduracion": [
            r"\b(?:ventana\s+de\s+maduraci[Ã³o]n)\b",
            r"\b(?:horizonte\s+(?:de\s+)?resultados?)\b",
            r"\b(?:rezago(?:s)?\s+(?:temporal(?:es)?|esperado(?:s)?))\b",
        ],
        "nivel_ambicion": [
            r"\b(?:nivel\s+de\s+ambici[Ã³o]n)\b",
            r"\b(?:metas?\s+(?:incrementales?|transformacionales?))\b",
        ],
    },
    CausalDimension.D5_IMPACTOS: {
        "efectos_largo_plazo": [
            r"\b(?:impacto(?:s)?\s+(?:esperado(?:s)?|de\s+largo\s+plazo))\b",
            r"\b(?:efectos?\s+(?:sostenidos?|duraderos?))\b",
            r"\b(?:transformaci[Ã³o]n\s+(?:estructural|sistÃ©mica))\b",
        ],
        "rutas_transmision": [
            r"\b(?:ruta(?:s)?\s+de\s+transmisi[Ã³o]n)\b",
            r"\b(?:canales?\s+(?:de\s+)?(?:impacto|propagaci[Ã³o]n))\b",
            r"\b(?:efectos?\s+(?:directos?|indirectos?|multiplicadores?))\b",
        ],
        "proxies_mensurables": [
            r"\b(?:proxies?\s+(?:de\s+)?impacto)\b",
            r"\b(?:indicadores?\s+(?:compuestos?|s[Ãi]ntesis))\b",
            r"\b(?:medidas?\s+(?:indirectas?|aproximadas?))\b",
        ],
        "alineacion_marcos": [
            r"\b(?:alineaci[Ã³o]n\s+con\s+(?:PND|Plan\s+Nacional))\b",
            r"\b(?:ODS\s+\d+|Objetivo(?:s)?\s+de\s+Desarrollo\s+Sostenible)\b",
            r"\b(?:coherencia\s+(?:vertical|horizontal))\b",
        ],
    },
    CausalDimension.D6_CAUSALIDAD: {
        "teoria_cambio_explicita": [
            r"\b(?:teor[Ãi]a\s+de(?:l)?\s+cambio)\b",
            r"\b(?:modelo\s+l[Ã³o]gico\s+(?:integrado|completo))\b",
            r"\b(?:marco\s+causal\s+(?:expl[Ãi]cito|formalizado))\b",
        ],
        "diagrama_causal": [
            r"\b(?:diagrama\s+(?:causal|DAG|de\s+flujo))\b",
            r"\b(?:representaci[Ã³o]n\s+gr[Ã¡a]fica\s+causal)\b",
            r"\b(?:mapa\s+(?:de\s+)?relaciones?)\b",
        ],
        "supuestos_verificables": [
            r"\b(?:supuestos?\s+(?:verificables?|cr[Ãi]ticos?))\b",
            r"\b(?:hip[Ã³o]tesis\s+(?:causales?|comprobables?))\b",
            r"\b(?:condiciones?\s+(?:necesarias?|suficientes?))\b",
```

```
6131:         ],
6132:         "mediadores_moderadores": [
6133:             r"\b(?:mediador(?:es)?|moderador(?:es)?)\b",
6134:             r"\b(?:variables?\s+(?:intermedias?|mediadoras?|moderadoras?))\b",
6135:         ],
6136:         "validacion_logica": [
6137:             r"\b(?:validaci[Ã³o]n\s+(?:l[Ã³o]gica|emp[Ã-i]rica))\b",
6138:             r"\b(?:pruebas?\s+(?:de\s+)?consistencia)\b",
6139:             r"\b(?:auditor[Ã-i]a\s+causal)\b",
6140:         ],
6141:         "sistema_seguimiento": [
6142:             r"\b(?:sistema\s+de\s+(?:seguimiento|monitoreo))\b",
6143:             r"\b(?:tablero\s+de\s+(?:control|indicadores))\b",
6144:             r"\b(?:evaluaci[Ã³o]n\s+(?:continua|peri[Ã³o]dica))\b",
6145:         ],
6146:     },
6147: }
6148:
6149: # ============================================================================
6150: # CONFIGURATION ARCHITECTURE
6151: # ============================================================================
6152:
6153: @dataclass(frozen=True)
6154: class ProcessorConfig:
6155:     """Immutable configuration for policy plan processing."""
6156:
6157:     preserve_document_structure: bool = True
6158:     enable_semantic_tagging: bool = True
6159:     confidence_threshold: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements",
 "auto_param_L301_34", 0.65)
6160:     context_window_chars: int = 400
6161:     max_evidence_per_pattern: int = 5
6162:     enable_bayesian_scoring: bool = True
6163:     utf8_normalization_form: str = "NFC"
6164:
6165:     # Advanced controls
6166:     entropy_weight: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto
_param_L308_28", 0.3)
6167:     proximity_decay_rate: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements",
 "auto_param_L309_34", 0.15)
6168:     min_sentence_length: int = 20
6169:     max_sentence_length: int = 500
6170:     bayesian_prior_confidence: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_stateme
nts", "auto_param_L312_39", 0.5)
6171:     bayesian_entropy_weight: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statement
s", "auto_param_L313_37", 0.3)
6172:     minimum_dimension_scores: dict[str, float] = field(
6173:         default_factory=lambda: {
6174:             "D1": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L31
6_18", 0.50),
6175:             "D2": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L31
7_18", 0.50),
6176:             "D3": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L31
8_18", 0.50),
6177:             "D4": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L31
9_18", 0.50),
```

```
 6178:             "D5": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L320_18", 0.50),
 6179:             "D6": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L321_18", 0.50),
 6180:         }
 6181:     )
 6182:     critical_dimension_overrides: dict[str, float] = field(
 6183:         default_factory=lambda: {"D1": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L325_39", 0.55), "D6": ParameterLoaderV2.get("farfan_core.processing.policy_processor._FallbackContradictionDetector._extract_policy_statements", "auto_param_L325_51", 0.55)}
 6184:     )
 6185:     differential_focus_indicators: tuple[str, ...] = (
 6186:         "enfoque diferencial",
 6187:         "enfoque de gÃ©nero",
 6188:         "mujeres rurales",
 6189:         "poblaciÃ³n vÃctima",
 6190:         "firmantes del acuerdo",
 6191:         "comunidades indÃgenas",
 6192:         "poblaciÃ³n LGBTIQ+",
 6193:         "juventud rural",
 6194:         "comunidades ribereÃ±as",
 6195:     )
 6196:     adaptability_indicators: tuple[str, ...] = (
 6197:         "mecanismo de ajuste",
 6198:         "retroalimentaciÃ³n",
 6199:         "aprendizaje",
 6200:         "monitoreo adaptativo",
 6201:         "ciclo de mejora",
 6202:         "sistema de alerta temprana",
 6203:         "evaluaciÃ³n continua",
 6204:     )
 6205:
 6206:     LEGACY_PARAM_MAP: ClassVar[dict[str, str]] = {
 6207:         "keep_structure": "preserve_document_structure",
 6208:         "tag_elements": "enable_semantic_tagging",
 6209:         "threshold": "confidence_threshold",
 6210:     }
 6211:
 6212:     @classmethod
 6213:     def from_legacy(cls, **kwargs: Any) -> "ProcessorConfig":
 6214:         """Construct configuration from legacy parameter names."""
 6215:         normalized = {}
 6216:         for key, value in kwargs.items():
 6217:             canonical = cls.LEGACY_PARAM_MAP.get(key, key)
 6218:             normalized[canonical] = value
 6219:         return cls(**normalized)
 6220:
 6221:     @calibrated_method("farfan_core.processing.policy_processor.ProcessorConfig.validate")
 6222:     def validate(self) -> None:
 6223:         """Validate configuration parameters."""
 6224:         if not ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L366_15", 0.0) <= self.confidence_threshold <= ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L366_51", 1.0):
 6225:             raise ValueError("confidence_threshold must be in [0, 1]")
 6226:         if self.context_window_chars < 100:
 6227:             raise ValueError("context_window_chars must be >= 100")
 6228:         if self.entropy_weight < 0 or self.entropy_weight > 1:
```

```
            raise ValueError("entropy_weight must be in [0, 1]")
        if not ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L372_15", 0.0) <= self.bayesian_prior_confidence <= ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L372_56", 1.0):
            raise ValueError("bayesian_prior_confidence must be in [0, 1]")
        if not ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L374_15", 0.0) <= self.bayesian_entropy_weight <= ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L374_54", 1.0):
            raise ValueError("bayesian_entropy_weight must be in [0, 1]")
        for dimension, threshold in self.minimum_dimension_scores.items():
            if not ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L377_19", 0.0) <= threshold <= ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L377_39", 1.0):
                raise ValueError(
                    f"minimum_dimension_scores[{dimension}] must be in [0, 1]"
                )
        for dimension, threshold in self.critical_dimension_overrides.items():
            if not ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L382_19", 0.0) <= threshold <= ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L382_39", 1.0):
                raise ValueError(
                    f"critical_dimension_overrides[{dimension}] must be in [0, 1]"
                )

# ============================================================================
# MATHEMATICAL SCORING ENGINE
# ============================================================================

class BayesianEvidenceScorer:
    """
    Bayesian evidence accumulation with entropy-weighted confidence scoring.

    Implements a modified Dempster-Shafer framework for multi-evidence fusion
    with automatic calibration against ground-truth policy corpora.
    """

    def __init__(
        self,
        prior_confidence: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L401_34", 0.5),
        entropy_weight: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L402_32", 0.3),
        calibration: dict[str, Any] | None = None,
    ) -> None:
        self.prior = prior_confidence
        self.entropy_weight = entropy_weight
        self._evidence_cache: dict[str, float] = {}
        self.calibration = calibration or {}

        # Defaults that can be overridden by calibration manifests
        self.epsilon_clip: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L411_35", 0.02)
        self.duplicate_gamma: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L412_38", 1.0)
        self.cross_type_floor: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L413_39", 0.0)
        self.source_quality_weights: dict[str, float] = {}
        self.sector_multipliers: dict[str, float] = {}
        self.sector_default: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L416_37", 1.0)
        self.municipio_multipliers: dict[str, float] = {}
        self.municipio_default: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.ProcessorConfig.validate", "auto_param_L418_40", 1.0)

        self._configure_from_calibration()

    @calibrated_method("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration")
```

```
    def _configure_from_calibration(self) -> None:
        config = self.calibration.get("bayesian_inference_robust") if isinstance(self.calibration, dict) else {}
        if not isinstance(config, dict):
            return

        evidence_cfg = config.get("mechanistic_evidence_system", {})
        if isinstance(evidence_cfg, dict):
            stability = evidence_cfg.get("stability_controls", {})
            if isinstance(stability, dict):
                self.epsilon_clip = float(stability.get("epsilon_clip", self.epsilon_clip))
                self.duplicate_gamma = float(stability.get("duplicate_gamma", self.duplicate_gamma))
                self.cross_type_floor = float(stability.get("cross_type_floor", self.cross_type_floor))
                self.epsilon_clip = min(max(self.epsilon_clip, ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._config
ure_from_calibration", "auto_param_L435_63", 0.0)), ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibrati
on", "auto_param_L435_69", 0.45))
                self.duplicate_gamma = max(ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration
", "auto_param_L436_43", 0.0), self.duplicate_gamma)
                self.cross_type_floor = max(ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibratio
n", "auto_param_L437_44", 0.0), min(ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param
_L437_53", 1.0), self.cross_type_floor))

            weights = evidence_cfg.get("source_quality_weights", {})
            if isinstance(weights, dict):
                self.source_quality_weights = {str(k): float(v) for k, v in weights.items() if isinstance(v, (int, float))}

        context_cfg = config.get("theoretically_grounded_priors", {})
        if isinstance(context_cfg, dict):
            hierarchy = context_cfg.get("hierarchical_context_priors", {})
            if isinstance(hierarchy, dict):
                sector = hierarchy.get("sector_multipliers", {})
                if isinstance(sector, dict):
                    self.sector_multipliers = {str(k).lower(): float(v) for k, v in sector.items() if isinstance(v, (int, float))}
                    self.sector_default = float(self.sector_multipliers.get("default", ParameterLoaderV2.get("farfan_core.processing.policy_processor.Bayesi
anEvidenceScorer._configure_from_calibration", "auto_param_L450_87", 1.0)))
                muni = hierarchy.get("municipio_tamano_multipliers", {})
                if isinstance(muni, dict):
                    self.municipio_multipliers = {str(k).lower(): float(v) for k, v in muni.items() if isinstance(v, (int, float))}
                    self.municipio_default = float(self.municipio_multipliers.get("default", ParameterLoaderV2.get("farfan_core.processing.policy_processor.
BayesianEvidenceScorer._configure_from_calibration", "auto_param_L454_93", 1.0)))

    def compute_evidence_score(
        self,
        matches: list[str],
        total_corpus_size: int,
        pattern_specificity: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "au
to_param_L460_37", 0.8),
        **kwargs: Any
    ) -> float:
        """
        Compute probabilistic confidence score for evidence matches.

        Args:
            matches: List of matched text segments
            total_corpus_size: Total document size in characters
            pattern_specificity: Pattern discrimination power [0,1]
            **kwargs: Additional optional parameters for compatibility
```

```
6329:
6330:         Returns:
6331:             Calibrated confidence score in [0, 1]
6332:         """
6333:         if not matches:
6334:             return ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L476_19",
0.0)
6335:
6336:         # Term frequency normalization
6337:         tf = len(matches) / max(1, total_corpus_size / 1000)
6338:         if self.cross_type_floor:
6339:             tf = max(self.cross_type_floor, tf)
6340:
6341:         # Entropy-based diversity penalty
6342:         match_lengths = np.array([len(m) for m in matches])
6343:         entropy = self._calculate_shannon_entropy(match_lengths)
6344:
6345:         # Bayesian update
6346:         clip_low = self.epsilon_clip
6347:         clip_high = ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L489_20"
, 1.0) - self.epsilon_clip
6348:         pattern_specificity = max(clip_low, min(clip_high, pattern_specificity))
6349:
6350:         likelihood = min(ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L49
2_25", 1.0), tf * pattern_specificity)
6351:         posterior = (likelihood * self.prior) / (
6352:             (likelihood * self.prior) + ((1 - likelihood) * (1 - self.prior))
6353:         )
6354:
6355:         # Entropy-weighted adjustment
6356:         final_score = (1 - self.entropy_weight) * posterior + self.entropy_weight * (
6357:             1 - entropy
6358:         )
6359:
6360:         # Apply duplicate penalty if provided by caller
6361:         if kwargs.get("duplicate_penalty"):
6362:             final_score *= self.duplicate_gamma
6363:
6364:         # Apply source quality weighting
6365:         if self.source_quality_weights:
6366:             source_quality = kwargs.get("source_quality")
6367:             if source_quality is not None:
6368:                 weight = self._lookup_weight(self.source_quality_weights, source_quality, default=ParameterLoaderV2.get("farfan_core.processing.policy_proce
ssor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L510_98", 1.0))
6369:                 final_score *= weight
6370:
6371:         # Context multipliers (sector / municipality)
6372:         sector = kwargs.get("sector") or kwargs.get("policy_sector")
6373:         if self.sector_multipliers:
6374:             final_score *= self._lookup_weight(self.sector_multipliers, sector, default=self.sector_default)
6375:
6376:         municipio = kwargs.get("municipio_tamano") or kwargs.get("municipio_size")
6377:         if self.municipio_multipliers:
6378:             final_score *= self._lookup_weight(self.municipio_multipliers, municipio, default=self.municipio_default)
6379:
6380:         return np.clip(final_score, ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "aut
```

```
o_param_L522_36", 0.0), ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L522_41", 1
.0))
 6381:
 6382:     @staticmethod
 6383:     def _calculate_shannon_entropy(values: np.ndarray, **kwargs: Any) -> float:
 6384:         """Calculate normalized Shannon entropy for value distribution.
 6385:
 6386:         Args:
 6387:             values: Array of numerical values
 6388:             **kwargs: Additional optional parameters for compatibility
 6389:
 6390:         Returns:
 6391:             Normalized Shannon entropy
 6392:         """
 6393:         if len(values) < 2:
 6394:             return ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration", "auto_param_L536_19",
 0.0)
 6395:
 6396:         # Discrete probability distribution
 6397:         hist, _ = np.histogram(values, bins=min(10, len(values)))
 6398:         prob = hist / hist.sum()
 6399:         prob = prob[prob > 0]  # Remove zeros
 6400:
 6401:         entropy = -np.sum(prob * np.log2(prob))
 6402:         max_entropy = np.log2(len(prob)) if len(prob) > 1 else ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._config
ure_from_calibration", "auto_param_L544_63", 1.0)
 6403:
 6404:         return entropy / max_entropy if max_entropy > 0 else ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidenceScorer._configur
e_from_calibration", "auto_param_L546_61", 0.0)
 6405:
 6406:     @staticmethod
 6407:     def _lookup_weight(mapping: dict[str, float], key: Any, default: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.BayesianEvidence
Scorer._configure_from_calibration", "auto_param_L549_77", 1.0)) -> float:
 6408:         if not mapping:
 6409:             return default
 6410:         if key is None:
 6411:             return mapping.get("default", default)
 6412:         if isinstance(key, str):
 6413:             direct = mapping.get(key)
 6414:             if direct is not None:
 6415:                 return direct
 6416:             lowered = key.lower()
 6417:             for candidate, value in mapping.items():
 6418:                 if isinstance(candidate, str) and candidate.lower() == lowered:
 6419:                     return value
 6420:         return mapping.get("default", default)
 6421:
 6422: # ============================================================================
 6423: # ADVANCED TEXT PROCESSOR
 6424: # ============================================================================
 6425:
 6426: class PolicyTextProcessor:
 6427:     """
 6428:     Industrial-grade text processing with multi-scale segmentation and
 6429:     coherence-preserving normalization for policy document analysis.
 6430:     """
```

```
6431:
6432:     def __init__(self, config: ProcessorConfig, *, calibration: dict[str, Any] | None = None) -> None:
6433:         self.config = config
6434:         self.calibration = calibration or {}
6435:         self._compiled_patterns: dict[str, re.Pattern] = {}
6436:         self._sentence_boundaries = re.compile(
6437:             r"(?<=[.!?])\s+(?=[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221])|(?<=\n\n)"
6438:         )
6439:
6440:     @calibrated_method("farfan_core.processing.policy_processor.PolicyTextProcessor.normalize_unicode")
6441:     def normalize_unicode(self, text: str) -> str:
6442:         """Apply canonical Unicode normalization (NFC/NFKC)."""
6443:         return unicodedata.normalize(self.config.utf8_normalization_form, text)
6444:
6445:     @calibrated_method("farfan_core.processing.policy_processor.PolicyTextProcessor.segment_into_sentences")
6446:     def segment_into_sentences(self, text: str, **kwargs: Any) -> list[str]:
6447:         """
6448:         Segment text into sentences with context-aware boundary detection.
6449:         Handles abbreviations, numerical lists, and Colombian naming conventions.
6450:
6451:         Args:
6452:             text: Input text to segment
6453:             **kwargs: Additional optional parameters for compatibility
6454:
6455:         Returns:
6456:             List of sentence strings
6457:         """
6458:         # Protect common abbreviations
6459:         protected = text
6460:         protected = re.sub(r"\bDr\.", "Dr___", protected)
6461:         protected = re.sub(r"\bSr\.", "Sr___", protected)
6462:         protected = re.sub(r"\bart\.", "art___", protected)
6463:         protected = re.sub(r"\bInc\.", "Inc___", protected)
6464:
6465:         sentences = self._sentence_boundaries.split(protected)
6466:
6467:         # Restore protected patterns
6468:         sentences = [s.replace("___", ".") for s in sentences]
6469:
6470:         # Filter by length constraints
6471:         return [
6472:             s.strip()
6473:             for s in sentences
6474:             if self.config.min_sentence_length
6475:             <= len(s.strip())
6476:             <= self.config.max_sentence_length
6477:         ]
6478:
6479:     def extract_contextual_window(
6480:         self, text: str, match_position: int, window_size: int
6481:     ) -> str:
6482:         """Extract semantically coherent context window around a match."""
6483:         start = max(0, match_position - window_size // 2)
6484:         end = min(len(text), match_position + window_size // 2)
6485:
6486:         # Expand to sentence boundaries
```

```
6487:         while start > 0 and text[start] not in ".!?\n":
6488:             start -= 1
6489:         while end < len(text) and text[end] not in ".!?\n":
6490:             end += 1
6491:
6492:         return text[start:end].strip()
6493:
6494:     @lru_cache(maxsize=256)
6495:     @calibrated_method("farfan_core.processing.policy_processor.PolicyTextProcessor.compile_pattern")
6496:     def compile_pattern(self, pattern_str: str) -> re.Pattern:
6497:         """Cache and compile regex patterns for performance."""
6498:         return re.compile(pattern_str, re.IGNORECASE | re.UNICODE)
6499:
6500: # ============================================================================
6501: # CORE INDUSTRIAL PROCESSOR
6502: # ============================================================================
6503:
6504: @dataclass
6505: class EvidenceBundle:
6506:     """Structured evidence container with provenance and confidence metadata."""
6507:
6508:     dimension: CausalDimension
6509:     category: str
6510:     matches: list[str] = field(default_factory=list)
6511:     confidence: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.PolicyTextProcessor.compile_pattern", "auto_param_L653_24", 0.0)
6512:     context_windows: list[str] = field(default_factory=list)
6513:     match_positions: list[int] = field(default_factory=list)
6514:
6515:     @calibrated_method("farfan_core.processing.policy_processor.EvidenceBundle.to_dict")
6516:     def to_dict(self) -> dict[str, Any]:
6517:         return {
6518:             "dimension": self.dimension.value,
6519:             "category": self.category,
6520:             "match_count": len(self.matches),
6521:             "confidence": round(self.confidence, 4),
6522:             "evidence_samples": self.matches[:3],
6523:             "context_preview": self.context_windows[:2],
6524:         }
6525:
6526: class IndustrialPolicyProcessor:
6527:     """
6528:     State-of-the-art policy plan processor implementing rigorous causal
6529:     framework analysis with Bayesian evidence scoring and graph-theoretic
6530:     validation for Colombian local development plans.
6531:
6532:     This processor provides core analysis capabilities for policy documents.
6533:
6534:     DEPRECATION NOTE: The questionnaire_path parameter is deprecated.
6535:     Modern pipelines use SPC (Smart Policy Chunks) ingestion which handles
6536:     questionnaire integration separately.
6537:     """
6538:
6539:     def __init__(
6540:         self,
6541:         config: ProcessorConfig | None = None,
6542:         questionnaire_path: Path | None = None,  # DEPRECATED: Kept for API compatibility only
```

```
        *,
        ontology: PortMunicipalOntology | None = None,
        semantic_analyzer: PortSemanticAnalyzer | None = None,
        performance_analyzer: PortPerformanceAnalyzer | None = None,
        contradiction_detector: PortContradictionDetector | None = None,
        temporal_verifier: PortTemporalLogicVerifier | None = None,
        confidence_calculator: PortBayesianConfidenceCalculator | None = None,
        municipal_analyzer: PortMunicipalAnalyzer | None = None,
    ) -> None:
        # DEPRECATION WARNING: questionnaire_path parameter is deprecated
        if questionnaire_path is not None:
            import warnings
            warnings.warn(
                "The 'questionnaire_path' parameter is deprecated and will be ignored. "
                "Modern SPC pipelines handle questionnaire integration separately. "
                "Use CPPIngestionPipeline instead.",
                DeprecationWarning,
                stacklevel=2
            )

        self.config = config or ProcessorConfig()
        self.config.validate()

        self.text_processor = PolicyTextProcessor(self.config)
        self.scorer = BayesianEvidenceScorer(
            prior_confidence=self.config.bayesian_prior_confidence,
            entropy_weight=self.config.bayesian_entropy_weight,
        )

        if ontology is None or semantic_analyzer is None or performance_analyzer is None:
            from farfan_pipeline.core.wiring.analysis_factory import (
                create_municipal_ontology,
                create_semantic_analyzer,
                create_performance_analyzer,
            )
            ontology = ontology or create_municipal_ontology()
            semantic_analyzer = semantic_analyzer or create_semantic_analyzer(ontology)
            performance_analyzer = performance_analyzer or create_performance_analyzer(ontology)

        if contradiction_detector is None:
            from farfan_pipeline.core.wiring.analysis_factory import create_contradiction_detector
            contradiction_detector = create_contradiction_detector()

        if temporal_verifier is None:
            from farfan_pipeline.core.wiring.analysis_factory import create_temporal_logic_verifier
            temporal_verifier = create_temporal_logic_verifier()

        if confidence_calculator is None:
            from farfan_pipeline.core.wiring.analysis_factory import create_bayesian_confidence_calculator
            confidence_calculator = create_bayesian_confidence_calculator()

        if municipal_analyzer is None:
            from farfan_pipeline.core.wiring.analysis_factory import create_municipal_analyzer
            municipal_analyzer = create_municipal_analyzer()

        self.ontology = ontology
```

```
6599:         self.semantic_analyzer = semantic_analyzer
6600:         self.performance_analyzer = performance_analyzer
6601:         self.contradiction_detector = contradiction_detector
6602:         self.temporal_verifier = temporal_verifier
6603:         self.confidence_calculator = confidence_calculator
6604:         self.municipal_analyzer = municipal_analyzer
6605:
6606:         # LEGACY: Questionnaire loading removed - this component is deprecated
6607:         # Modern SPC pipeline handles questionnaire injection separately
6608:         self.questionnaire_file_path = None
6609:         self.questionnaire_data = {"questions": []}  # Empty stub for backward compatibility
6610:
6611:         # Compile pattern taxonomy
6612:         self._pattern_registry = self._compile_pattern_registry()
6613:
6614:         # Policy point keyword extraction
6615:         self.point_patterns: dict[str, re.Pattern] = {}
6616:         self._build_point_patterns()
6617:
6618:         # Processing statistics
6619:         self.statistics: dict[str, Any] = defaultdict(int)
6620:
6621:     @calibrated_method("farfan_core.processing.policy_processor.IndustrialPolicyProcessor._load_questionnaire")
6622:     def _load_questionnaire(self) -> dict[str, Any]:
6623:         """
6624:         LEGACY: Questionnaire loading disabled.
6625:
6626:         This method is kept for backward compatibility but returns empty data.
6627:         Modern SPC pipeline handles questionnaire injection separately.
6628:         """
6629:         logger.warning(
6630:             "IndustrialPolicyProcessor._load_questionnaire called but questionnaire "
6631:             "loading is disabled. This is a legacy component. Use SPC ingestion instead."
6632:         )
6633:         return {"questions": []}
6634:
6635:     @calibrated_method("farfan_core.processing.policy_processor.IndustrialPolicyProcessor._compile_pattern_registry")
6636:     def _compile_pattern_registry(self) -> dict[CausalDimension, dict[str, list[re.Pattern]]]:
6637:         """Compile all causal patterns into efficient regex objects."""
6638:         registry = {}
6639:         for dimension, categories in CAUSAL_PATTERN_TAXONOMY.items():
6640:             registry[dimension] = {}
6641:             for category, patterns in categories.items():
6642:                 registry[dimension][category] = [
6643:                     self.text_processor.compile_pattern(p) for p in patterns
6644:                 ]
6645:         return registry
6646:
6647:     @calibrated_method("farfan_core.processing.policy_processor.IndustrialPolicyProcessor._build_point_patterns")
6648:     def _build_point_patterns(self) -> None:
6649:         """
6650:         LEGACY: Pattern building from questionnaire disabled.
6651:
6652:         This method is kept for backward compatibility but does nothing.
6653:         Modern SPC pipeline handles question-aware chunking separately.
6654:         """
```

```
6655:         questions = self.questionnaire_data.get("questions", [])
6656:
6657:         if not questions:
6658:             logger.info(
6659:                 "No questionnaire questions available. "
6660:                 "This is expected for legacy IndustrialPolicyProcessor. "
6661:                 "Use SPC ingestion for question-aware analysis."
6662:             )
6663:             return
6664:
6665:         # Legacy path (should not be reached in modern pipeline)
6666:         point_keywords: dict[str, set[str]] = defaultdict(set)
6667:
6668:         for question in questions:
6669:             point_code = question.get("point_code")
6670:             if not point_code:
6671:                 continue
6672:
6673:             # Extract title keywords
6674:             title = question.get("point_title", "").lower()
6675:             if title:
6676:                 point_keywords[point_code].add(title)
6677:
6678:             # Extract hint keywords (cleaned)
6679:             for hint in question.get("hints", []):
6680:                 cleaned = re.sub(r"[()]", "", hint).strip().lower()
6681:                 if len(cleaned) > 3:
6682:                     point_keywords[point_code].add(cleaned)
6683:
6684:         # Compile into optimized regex patterns
6685:         for point_code, keywords in point_keywords.items():
6686:             # Sort by length (prioritize longer phrases)
6687:             sorted_kw = sorted(keywords, key=len, reverse=True)
6688:             pattern_str = "|".join(rf"\b{re.escape(kw)}\b" for kw in sorted_kw if kw)
6689:             self.point_patterns[point_code] = re.compile(pattern_str, re.IGNORECASE)
6690:
6691:         logger.info(f"Compiled patterns for {len(self.point_patterns)} policy points")
6692:
6693:     @calibrated_method("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process")
6694:     def process(self, raw_text: str, **kwargs: Any) -> dict[str, Any]:
6695:         """
6696:         Execute comprehensive policy plan analysis.
6697:
6698:         Args:
6699:             raw_text: Sanitized policy document text
6700:             **kwargs: Additional optional parameters (e.g., text, sentences, tables) for compatibility
6701:
6702:         Returns:
6703:             Structured analysis results with evidence bundles and confidence scores
6704:         """
6705:         if not raw_text or len(raw_text) < 100:
6706:             logger.warning("Input text too short for analysis")
6707:             return self._empty_result()
6708:
6709:         # Normalize and segment
6710:         normalized = self.text_processor.normalize_unicode(raw_text)
```

```
        sentences = self.text_processor.segment_into_sentences(normalized)

        logger.info(f"Processing document: {len(normalized)} chars, {len(sentences)} sentences")

        # Extract metadata
        metadata = self._extract_metadata(normalized)

        # Evidence extraction by policy point
        point_evidence = {}
        for point_code in sorted(self.point_patterns.keys()):
            evidence = self._extract_point_evidence(
                normalized, sentences, point_code
            )
            if evidence:
                point_evidence[point_code] = evidence

        # Global causal dimension analysis
        dimension_analysis = self._analyze_causal_dimensions(normalized, sentences)

        # Semantic diagnostics and performance evaluation
        semantic_cube = self.semantic_analyzer.extract_semantic_cube(sentences)
        performance_analysis = self.performance_analyzer.analyze_performance(
            semantic_cube
        )

        try:
            contradiction_bundle = self._run_contradiction_analysis(normalized, metadata)
        except PDETAnalysisException as exc:
            logger.error("Contradiction analysis failed: %s", exc)
            contradiction_bundle = {
                "reports": {},
                "temporal_assessments": {},
                "bayesian_scores": {},
                "critical_diagnosis": {
                    "critical_links": {},
                    "risk_assessment": {},
                    "intervention_recommendations": {},
                },
            }

        quality_score = self._calculate_quality_score(
            dimension_analysis, contradiction_bundle, performance_analysis
        )

        summary = self.municipal_analyzer._generate_summary(
            semantic_cube,
            performance_analysis,
            contradiction_bundle["critical_diagnosis"],
        )

        # Compile results
        return {
            "metadata": metadata,
            "point_evidence": point_evidence,
            "dimension_analysis": dimension_analysis,
            "semantic_cube": semantic_cube,
```

```
            "performance_analysis": performance_analysis,
            "critical_diagnosis": contradiction_bundle["critical_diagnosis"],
            "contradiction_reports": contradiction_bundle["reports"],
            "temporal_consistency": contradiction_bundle["temporal_assessments"],
            "bayesian_dimension_scores": contradiction_bundle["bayesian_scores"],
            "quality_score": asdict(quality_score),
            "summary": summary,
            "document_statistics": {
                "character_count": len(normalized),
                "sentence_count": len(sentences),
                "point_coverage": len(point_evidence),
                "avg_confidence": self._compute_avg_confidence(dimension_analysis),
            },
            "processing_status": "complete",
            "config_snapshot": {
                "confidence_threshold": self.config.confidence_threshold,
                "bayesian_enabled": self.config.enable_bayesian_scoring,
            },
        }

    def _match_patterns_in_sentences(
        self, compiled_patterns: list, relevant_sentences: list[str], **kwargs: Any
    ) -> tuple[list[str], list[int]]:
        """
        Execute pattern matching across relevant sentences and collect matches with positions.

        Args:
            compiled_patterns: List of compiled regex patterns to match
            relevant_sentences: Filtered sentences to search within
            **kwargs: Additional optional parameters for compatibility

        Returns:
            Tuple of (matched_strings, match_positions)
        """
        matches = []
        positions = []

        for compiled_pattern in compiled_patterns:
            for sentence in relevant_sentences:
                for match in compiled_pattern.finditer(sentence):
                    matches.append(match.group(0))
                    positions.append(match.start())

        return matches, positions

    def _compute_evidence_confidence(
        self, matches: list[str], text_length: int, pattern_specificity: float, **kwargs: Any
    ) -> float:
        """
        Calculate confidence score for evidence based on pattern matches and contextual factors.

        Args:
            matches: List of matched pattern strings
            text_length: Total length of the document text
            pattern_specificity: Specificity coefficient for pattern weighting
            **kwargs: Additional optional parameters for compatibility
```

```
6823:
6824:         Returns:
6825:             Computed confidence score
6826:         """
6827:         confidence = self.scorer.compute_evidence_score(
6828:             matches, text_length, pattern_specificity=pattern_specificity
6829:         )
6830:         return confidence
6831:
6832:     def _construct_evidence_bundle(
6833:         self,
6834:         dimension: CausalDimension,
6835:         category: str,
6836:         matches: list[str],
6837:         positions: list[int],
6838:         confidence: float,
6839:         **kwargs: Any
6840:     ) -> dict[str, Any]:
6841:         """
6842:         Assemble evidence bundle from matched patterns and computed confidence.
6843:
6844:         Args:
6845:             dimension: Causal dimension classification
6846:             category: Specific category within dimension
6847:             matches: List of matched pattern strings
6848:             positions: List of match positions in text
6849:             confidence: Computed confidence score
6850:             **kwargs: Additional optional parameters for compatibility
6851:
6852:         Returns:
6853:             Serialized evidence bundle dictionary
6854:         """
6855:         bundle = EvidenceBundle(
6856:             dimension=dimension,
6857:             category=category,
6858:             matches=matches[: self.config.max_evidence_per_pattern],
6859:             confidence=confidence,
6860:             match_positions=positions[: self.config.max_evidence_per_pattern],
6861:         )
6862:         return bundle.to_dict()
6863:
6864:     def _run_contradiction_analysis(
6865:         self, text: str, metadata: dict[str, Any]
6866:     ) -> dict[str, Any]:
6867:         """Execute contradiction and temporal diagnostics across all dimensions."""
6868:
6869:         if not self.contradiction_detector:
6870:             raise PDETAnalysisException("Contradiction detector unavailable")
6871:
6872:         plan_name = metadata.get("title", "Plan de Desarrollo")
6873:         dimension_mapping = {
6874:             CausalDimension.D1_INSUMOS: ContradictionPolicyDimension.DIAGNOSTICO,
6875:             CausalDimension.D2_ACTIVIDADES: ContradictionPolicyDimension.ESTRATEGICO,
6876:             CausalDimension.D3_PRODUCTOS: ContradictionPolicyDimension.PROGRAMATICO,
6877:             CausalDimension.D4_RESULTADOS: ContradictionPolicyDimension.SEGUIMIENTO,
6878:             CausalDimension.D5_IMPACTOS: ContradictionPolicyDimension.TERRITORIAL,
```

```
 6879:             CausalDimension.D6_CAUSALIDAD: ContradictionPolicyDimension.ESTRATEGICO,
 6880:         }
 6881:
 6882:         domain_weights = {
 6883:             CausalDimension.D1_INSUMOS: 1.1,
 6884:             CausalDimension.D2_ACTIVIDADES: ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L
1000_44", 1.0),
 6885:             CausalDimension.D3_PRODUCTOS: ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L10
01_42", 1.0),
 6886:             CausalDimension.D4_RESULTADOS: 1.1,
 6887:             CausalDimension.D5_IMPACTOS: 1.15,
 6888:             CausalDimension.D6_CAUSALIDAD: 1.2,
 6889:         }
 6890:
 6891:         reports: dict[str, Any] = {}
 6892:         temporal_assessments: dict[str, Any] = {}
 6893:         bayesian_scores: dict[str, float] = {}
 6894:         critical_links: dict[str, Any] = {}
 6895:         risk_assessment: dict[str, Any] = {}
 6896:         intervention_recommendations: dict[str, Any] = {}
 6897:
 6898:         for dimension in CausalDimension:
 6899:             policy_dimension = dimension_mapping.get(dimension)
 6900:             try:
 6901:                 report = self.contradiction_detector.detect(
 6902:                     text, plan_name=plan_name, dimension=policy_dimension
 6903:                 )
 6904:             except Exception as exc:  # pragma: no cover - external deps
 6905:                 raise PDETAnalysisException(
 6906:                     f"Contradiction detection failed for {dimension.name}: {exc}"
 6907:                 ) from exc
 6908:
 6909:             reports[dimension.value] = report
 6910:
 6911:             try:
 6912:                 statements = self.contradiction_detector._extract_policy_statements(  # type: ignore[attr-defined]
 6913:                     text, policy_dimension
 6914:                 )
 6915:             except Exception:  # pragma: no cover - best effort if detector lacks method
 6916:                 statements = []
 6917:
 6918:             is_consistent, conflicts = self.temporal_verifier.verify_temporal_consistency(
 6919:                 statements
 6920:             )
 6921:             temporal_assessments[dimension.value] = {
 6922:                 "is_consistent": is_consistent,
 6923:                 "conflicts": conflicts,
 6924:             }
 6925:
 6926:             coherence_metrics = report.get("coherence_metrics", {})
 6927:             coherence_score = float(coherence_metrics.get("coherence_score", ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicy
Processor.process", "auto_param_L1043_77", 0.0)))
 6928:             observations = max(1, len(statements))
 6929:             posterior = self.confidence_calculator.calculate_posterior(
 6930:                 evidence_strength=max(coherence_score, ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "a
uto_param_L1046_55", 0.01)),
```

```
 6931:                 observations=observations,
 6932:                 domain_weight=domain_weights.get(dimension, ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process
", "auto_param_L1048_60", 1.0)),
 6933:             )
 6934:             bayesian_scores[dimension.value] = float(posterior)
 6935:
 6936:             total_contradictions = int(report.get("total_contradictions", 0))
 6937:             if total_contradictions:
 6938:                 keywords = []
 6939:                 # Defensive: ensure contradictions is a list
 6940:                 contradictions_list = ensure_list_return(report.get("contradictions", []))
 6941:                 for contradiction in contradictions_list:
 6942:                     ctype = contradiction.get("contradiction_type")
 6943:                     if ctype:
 6944:                         keywords.append(ctype)
 6945:
 6946:                 severity = 1 - coherence_score if coherence_score else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProces
sor.process", "auto_param_L1062_71", 0.5)
 6947:                 critical_links[dimension.value] = {
 6948:                     "criticality_score": round(min(ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_
param_L1064_51", 1.0), max(ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1064_60", 0.0), severity
)), 4),
 6949:                     "text_analysis": {
 6950:                         "sentiment": "negative" if coherence_score < ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcesso
r.process", "auto_param_L1066_69", 0.5) else "neutral",
 6951:                         "keywords": keywords,
 6952:                         "word_count": len(text.split()),
 6953:                     },
 6954:                 }
 6955:                 risk_assessment[dimension.value] = {
 6956:                     "overall_risk": "high" if total_contradictions > 3 else "medium",
 6957:                     "risk_factors": keywords,
 6958:                 }
 6959:                 intervention_recommendations[dimension.value] = report.get(
 6960:                     "recommendations", []
 6961:                 )
 6962:
 6963:         return {
 6964:             "reports": reports,
 6965:             "temporal_assessments": temporal_assessments,
 6966:             "bayesian_scores": bayesian_scores,
 6967:             "critical_diagnosis": {
 6968:                 "critical_links": critical_links,
 6969:                 "risk_assessment": risk_assessment,
 6970:                 "intervention_recommendations": intervention_recommendations,
 6971:             },
 6972:         }
 6973:
 6974:     def _calculate_quality_score(
 6975:         self,
 6976:         dimension_analysis: dict[str, Any],
 6977:         contradiction_bundle: dict[str, Any],
 6978:         performance_analysis: dict[str, Any],
 6979:     ) -> QualityScore:
 6980:         """Aggregate key indicators into a structured QualityScore dataclass."""
 6981:
```

```
 6982:         bayesian_scores = contradiction_bundle.get("bayesian_scores", {})
 6983:         bayesian_values = list(bayesian_scores.values())
 6984:         overall_score = float(np.mean(bayesian_values)) if bayesian_values else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPol
icyProcessor.process", "auto_param_L1100_80", 0.0)
 6985:
 6986:         def _dimension_confidence(key: CausalDimension) -> float:
 6987:             return float(
 6988:                 dimension_analysis.get(key.value, {}).get("dimension_confidence", ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialP
olicyProcessor.process", "auto_param_L1104_82", 0.0))
 6989:             )
 6990:
 6991:         temporal_flags = contradiction_bundle.get("temporal_assessments", {})
 6992:         temporal_values = [
 6993:             ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1109_12", 1.0) if assessment.get
("is_consistent", True) else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1109_62", 0.0)
 6994:             for assessment in temporal_flags.values()
 6995:         ]
 6996:         temporal_consistency = (
 6997:             float(np.mean(temporal_values)) if temporal_values else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor
.process", "auto_param_L1113_68", 1.0)
 6998:         )
 6999:
 7000:         reports = contradiction_bundle.get("reports", {})
 7001:         coherence_scores = [
 7002:             float(report.get("coherence_metrics", {}).get("coherence_score", ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicy
Processor.process", "auto_param_L1118_77", 0.0)))
 7003:             for report in reports.values()
 7004:         ]
 7005:         causal_coherence = float(np.mean(coherence_scores)) if coherence_scores else ParameterLoaderV2.get("farfan_core.processing.policy_processor.Industri
alPolicyProcessor.process", "auto_param_L1121_85", 0.0)
 7006:
 7007:         objective_alignment = float(
 7008:             reports.get(
 7009:                 CausalDimension.D4_RESULTADOS.value,
 7010:                 {},
 7011:             )
 7012:             .get("coherence_metrics", {})
 7013:             .get("objective_alignment", ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1129
_40", 0.0))
 7014:         )
 7015:
 7016:         confidence_interval = (
 7017:             float(min(bayesian_values)) if bayesian_values else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.pro
cess", "auto_param_L1133_64", 0.0),
 7018:             float(max(bayesian_values)) if bayesian_values else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.pro
cess", "auto_param_L1134_64", 0.0),
 7019:         )
 7020:
 7021:         evidence = {
 7022:             "bayesian_scores": bayesian_scores,
 7023:             "dimension_confidences": {
 7024:                 key: value.get("dimension_confidence", ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "a
uto_param_L1140_55", 0.0))
 7025:                 for key, value in dimension_analysis.items()
 7026:             },
 7027:             "performance_metrics": performance_analysis.get("value_chain_metrics", {}),
```

```
7028:         }
7029:
7030:         return QualityScore(
7031:             overall_score=overall_score,
7032:             financial_feasibility=_dimension_confidence(CausalDimension.D1_INSUMOS),
7033:             indicator_quality=_dimension_confidence(CausalDimension.D3_PRODUCTOS),
7034:             responsibility_clarity=_dimension_confidence(CausalDimension.D2_ACTIVIDADES),
7035:             temporal_consistency=temporal_consistency,
7036:             pdet_alignment=objective_alignment,
7037:             causal_coherence=causal_coherence,
7038:             confidence_interval=confidence_interval,
7039:             evidence=evidence,
7040:         )
7041:
7042:     def _extract_point_evidence(
7043:         self, text: str, sentences: list[str], point_code: str
7044:     ) -> dict[str, Any]:
7045:         """Extract evidence for a specific policy point across all dimensions."""
7046:         pattern = self.point_patterns.get(point_code)
7047:         if not pattern:
7048:             return {}
7049:
7050:         # Find relevant sentences
7051:         relevant_sentences = [s for s in sentences if pattern.search(s)]
7052:         if not relevant_sentences:
7053:             return {}
7054:
7055:         # Search for dimensional evidence within relevant context
7056:         evidence_by_dimension = {}
7057:         for dimension, categories in self._pattern_registry.items():
7058:             dimension_evidence = []
7059:
7060:             for category, compiled_patterns in categories.items():
7061:                 matches, positions = self._match_patterns_in_sentences(
7062:                     compiled_patterns, relevant_sentences
7063:                 )
7064:
7065:                 if matches:
7066:                     confidence = self._compute_evidence_confidence(
7067:                         matches, len(text), pattern_specificity=ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.pro
cess", "auto_param_L1183_64", 0.85)
7068:                     )
7069:
7070:                     if confidence >= self.config.confidence_threshold:
7071:                         evidence_dict = self._construct_evidence_bundle(
7072:                             dimension, category, matches, positions, confidence
7073:                         )
7074:                         dimension_evidence.append(evidence_dict)
7075:
7076:             if dimension_evidence:
7077:                 evidence_by_dimension[dimension.value] = dimension_evidence
7078:
7079:         return evidence_by_dimension
7080:
7081:     def _analyze_causal_dimensions(
7082:         self, text: str, sentences: list[str] | None = None
```

```
7083:     ) -> dict[str, Any]:
7084:         """
7085:         Perform global analysis of causal dimensions across entire document.
7086:
7087:         Args:
7088:             text: Full document text
7089:             sentences: Optional pre-segmented sentences. If not provided, will be
7090:                       automatically extracted from text using the text processor.
7091:
7092:         Returns:
7093:             Dictionary containing dimension scores and confidence metrics
7094:
7095:         Note:
7096:             This function requires 'sentences' for optimal performance. If not provided,
7097:             sentences will be extracted from text automatically, which may impact performance.
7098:         """
7099:         # Defensive validation: ensure sentences parameter is provided
7100:         if sentences is None:
7101:             logger.warning(
7102:                 "_analyze_causal_dimensions called without 'sentences' parameter. "
7103:                 "Automatically extracting sentences from text. "
7104:                 "Expected signature: _analyze_causal_dimensions(self, text: str, sentences: List[str])"
7105:             )
7106:             # Auto-extract sentences if not provided
7107:             sentences = self.text_processor.segment_into_sentences(text)
7108:
7109:         dimension_scores = {}
7110:
7111:         for dimension, categories in self._pattern_registry.items():
7112:             total_matches = 0
7113:             category_results = {}
7114:
7115:             for category, compiled_patterns in categories.items():
7116:                 matches = []
7117:                 for pattern in compiled_patterns:
7118:                     for sentence in sentences:
7119:                         matches.extend(pattern.findall(sentence))
7120:
7121:                 if matches:
7122:                     confidence = self.scorer.compute_evidence_score(
7123:                         matches, len(text), pattern_specificity=ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.pro
cess", "auto_param_L1239_64", 0.80)
7124:                     )
7125:                     category_results[category] = {
7126:                         "match_count": len(matches),
7127:                         "confidence": round(confidence, 4),
7128:                     }
7129:                     total_matches += len(matches)
7130:
7131:             dimension_scores[dimension.value] = {
7132:                 "categories": category_results,
7133:                 "total_matches": total_matches,
7134:                 "dimension_confidence": round(
7135:                     np.mean([c["confidence"] for c in category_results.values()])
7136:                     if category_results
7137:                     else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1253_25", 0.0),
```

```
                    4,
                ),
            }

        return dimension_scores

    @staticmethod
    def _extract_metadata(text: str) -> dict[str, Any]:
        """Extract key metadata from policy document header."""
        # Title extraction
        title_match = re.search(
            r"(?i)plan\s+(?:de\s+)?desarrollo\s+(?:municipal|departamental|local)?\s*[:\-]?\s*([^\n]{10,150})",
            text[:2000],
        )
        title = title_match.group(1).strip() if title_match else "Sin tÃtulo identificado"

        # Entity extraction
        entity_match = re.search(
            r"(?i)(?:municipio|alcald[Ãi]a|gobernaci[Ã³o]n|distrito)\s+(?:de\s+)?([A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©ÃÃ³ÃºÃ±\s]+)",
            text[:3000],
        )
        entity = entity_match.group(1).strip() if entity_match else "Entidad no especificada"

        # Period extraction
        period_match = re.search(r"(20\d{2})\s*[-â\200\223â\200\224]\s*(20\d{2})", text[:3000])
        period = {
            "start_year": int(period_match.group(1)) if period_match else None,
            "end_year": int(period_match.group(2)) if period_match else None,
        }

        return {
            "title": title,
            "entity": entity,
            "period": period,
            "extraction_timestamp": "2025-10-13",
        }

    @staticmethod
    def _compute_avg_confidence(dimension_analysis: dict[str, Any]) -> float:
        """Calculate average confidence across all dimensions."""
        confidences = [
            dim_data["dimension_confidence"]
            for dim_data in dimension_analysis.values()
            if dim_data.get("dimension_confidence", 0) > 0
        ]
        return round(np.mean(confidences), 4) if confidences else ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor.process", "auto_param_L1299_66", 0.0)

    @calibrated_method("farfan_core.processing.policy_processor.IndustrialPolicyProcessor._empty_result")
    def _empty_result(self) -> dict[str, Any]:
        """Return structure for failed/empty processing."""
        return {
            "metadata": {},
            "point_evidence": {},
            "dimension_analysis": {},
            "document_statistics": {
```

```
                "character_count": 0,
                "sentence_count": 0,
                "point_coverage": 0,
                "avg_confidence": ParameterLoaderV2.get("farfan_core.processing.policy_processor.IndustrialPolicyProcessor._empty_result", "auto_param_L1312
_34", 0.0),
            },
            "processing_status": "failed",
            "error": "Insufficient input for analysis",
        }

    def export_results(
        self, results: dict[str, Any], output_path: str | Path
    ) -> None:
        """Export analysis results to JSON with formatted output."""
        # Delegate to factory for I/O operation
        from farfan_pipeline.processing.factory import save_json

        save_json(results, output_path)
        logger.info(f"Results exported to {output_path}")

# ============================================================================
# ENHANCED SANITIZER WITH STRUCTURE PRESERVATION
# ============================================================================

class AdvancedTextSanitizer:
    """
    Sophisticated text sanitization preserving semantic structure and
    critical policy elements with differential privacy guarantees.
    """

    def __init__(self, config: ProcessorConfig) -> None:
        self.config = config
        self.protection_markers: dict[str, tuple[str, str]] = {
            "heading": ("__HEAD_START__", "__HEAD_END__"),
            "list_item": ("__LIST_START__", "__LIST_END__"),
            "table_cell": ("__TABLE_START__", "__TABLE_END__"),
            "citation": ("__CITE_START__", "__CITE_END__"),
        }

    @calibrated_method("farfan_core.processing.policy_processor.AdvancedTextSanitizer.sanitize")
    def sanitize(self, raw_text: str) -> str:
        """
        Execute comprehensive text sanitization pipeline.

        Pipeline stages:
        1. Unicode normalization (NFC)
        2. Structure element protection
        3. Whitespace normalization
        4. Special character handling
        5. Encoding validation
        """
        if not raw_text:
            return ""

        # Stage 1: Unicode normalization
        text = unicodedata.normalize(self.config.utf8_normalization_form, raw_text)
```

```
7248:
7249:         # Stage 2: Protect structural elements
7250:         if self.config.preserve_document_structure:
7251:             text = self._protect_structure(text)
7252:
7253:         # Stage 3: Whitespace normalization
7254:         text = re.sub(r"[ \t]+", " ", text)
7255:         text = re.sub(r"\n{3,}", "\n\n", text)
7256:
7257:         # Stage 4: Remove control characters (except newlines/tabs)
7258:         text = "".join(
7259:             char for char in text
7260:             if unicodedata.category(char)[0] != "C" or char in "\n\t"
7261:         )
7262:
7263:         # Stage 5: Restore protected elements
7264:         if self.config.preserve_document_structure:
7265:             text = self._restore_structure(text)
7266:
7267:         return text.strip()
7268:
7269:     @calibrated_method("farfan_core.processing.policy_processor.AdvancedTextSanitizer._protect_structure")
7270:     def _protect_structure(self, text: str) -> str:
7271:         """Mark structural elements for protection during sanitization."""
7272:         protected = text
7273:
7274:         # Protect headings (numbered or capitalized lines)
7275:         heading_pattern = re.compile(
7276:             r"^(?:[\d.]+\s+)?([A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221a-zÃ¡Ã©Ã-Ã³Ãºñ\s]{5,80})$",
7277:             re.MULTILINE,
7278:         )
7279:         for match in reversed(list(heading_pattern.finditer(protected))):
7280:             start, end = match.span()
7281:             heading_text = match.group(0)
7282:             protected = (
7283:                 protected[:start]
7284:                 + f"{self.protection_markers['heading'][0]}{heading_text}{self.protection_markers['heading'][1]}"
7285:                 + protected[end:]
7286:             )
7287:
7288:         # Protect list items
7289:         list_pattern = re.compile(r"^[\s]*[â\200¢\-\*\d]+[\.\)]\s+(.+)$", re.MULTILINE)
7290:         for match in reversed(list(list_pattern.finditer(protected))):
7291:             start, end = match.span()
7292:             item_text = match.group(0)
7293:             protected = (
7294:                 protected[:start]
7295:                 + f"{self.protection_markers['list_item'][0]}{item_text}{self.protection_markers['list_item'][1]}"
7296:                 + protected[end:]
7297:             )
7298:
7299:         return protected
7300:
7301:     @calibrated_method("farfan_core.processing.policy_processor.AdvancedTextSanitizer._restore_structure")
7302:     def _restore_structure(self, text: str) -> str:
7303:         """Remove protection markers after sanitization."""
```

```
7304:         restored = text
7305:         for _marker_type, (start_mark, end_mark) in self.protection_markers.items():
7306:             restored = restored.replace(start_mark, "")
7307:             restored = restored.replace(end_mark, "")
7308:         return restored
7309:
7310: # ==========================================================================
7311: # INTEGRATED FILE HANDLING WITH RESILIENCE
7312: # ==========================================================================
7313:
7314: class ResilientFileHandler:
7315:     """
7316:     Production-grade file I/O with automatic encoding detection,
7317:     retry logic, and comprehensive error classification.
7318:     """
7319:
7320:     ENCODINGS = ["utf-8", "utf-8-sig", "latin-1", "cp1252", "iso-8859-1"]
7321:
7322:     @classmethod
7323:     def read_text(cls, file_path: str | Path) -> str:
7324:         """
7325:         Read text file with automatic encoding detection and fallback cascade.
7326:
7327:         Args:
7328:             file_path: Path to input file
7329:
7330:         Returns:
7331:             Decoded text content
7332:
7333:         Raises:
7334:             IOError: If file cannot be read with any supported encoding
7335:         """
7336:         # Delegate to factory for I/O operation
7337:         from farfan_pipeline.processing.factory import read_text_file
7338:
7339:         try:
7340:             return read_text_file(file_path, encodings=list(cls.ENCODINGS))
7341:         except Exception as e:
7342:             raise OSError(f"Failed to read {file_path} with any supported encoding") from e
7343:
7344:     @classmethod
7345:     def write_text(cls, content: str, file_path: str | Path) -> None:
7346:         """Write text content with UTF-8 encoding and directory creation."""
7347:         # Delegate to factory for I/O operation
7348:         from farfan_pipeline.processing.factory import write_text_file
7349:
7350:         write_text_file(content, file_path)
7351:
7352: # ==========================================================================
7353: # UNIFIED ORCHESTRATOR
7354: # ==========================================================================
7355:
7356: class PolicyAnalysisPipeline:
7357:     """
7358:     End-to-end orchestrator for Colombian local development plan analysis
7359:     implementing the complete DECALOGO causal framework evaluation workflow.
```

```
7360:
7361:     DEPRECATION NOTE: The questionnaire_path parameter is deprecated.
7362:     Modern pipelines use SPC (Smart Policy Chunks) ingestion which handles
7363:     questionnaire integration separately.
7364:     """
7365:
7366:     def __init__(
7367:         self,
7368:         config: ProcessorConfig | None = None,
7369:         questionnaire_path: Path | None = None,  # DEPRECATED: Kept for API compatibility only
7370:     ) -> None:
7371:         # DEPRECATION WARNING: questionnaire_path parameter is deprecated
7372:         if questionnaire_path is not None:
7373:             import warnings
7374:             warnings.warn(
7375:                 "The 'questionnaire_path' parameter is deprecated and will be ignored. "
7376:                 "Modern SPC pipelines handle questionnaire integration separately. "
7377:                 "Use CPPIngestionPipeline instead.",
7378:                 DeprecationWarning,
7379:                 stacklevel=2
7380:             )
7381:
7382:         self.config = config or ProcessorConfig()
7383:         self.sanitizer = AdvancedTextSanitizer(self.config)
7384:
7385:         from farfan_pipeline.core.wiring.analysis_factory import create_analysis_components
7386:
7387:         components = create_analysis_components()
7388:         self.document_loader = components['document_loader']
7389:         self.ontology = components['ontology']
7390:         self.semantic_analyzer = components['semantic_analyzer']
7391:         self.performance_analyzer = components['performance_analyzer']
7392:         self.temporal_verifier = components['temporal_verifier']
7393:         self.confidence_calculator = components['confidence_calculator']
7394:         self.contradiction_detector = components['contradiction_detector']
7395:         self.municipal_analyzer = components['municipal_analyzer']
7396:
7397:         self.processor = IndustrialPolicyProcessor(
7398:             self.config,
7399:             questionnaire_path,
7400:             ontology=self.ontology,
7401:             semantic_analyzer=self.semantic_analyzer,
7402:             performance_analyzer=self.performance_analyzer,
7403:             contradiction_detector=self.contradiction_detector,
7404:             temporal_verifier=self.temporal_verifier,
7405:             confidence_calculator=self.confidence_calculator,
7406:             municipal_analyzer=self.municipal_analyzer,
7407:         )
7408:         self.file_handler = ResilientFileHandler()
7409:
7410:     def analyze_file(
7411:         self,
7412:         input_path: str | Path,
7413:         output_path: str | Path | None = None,
7414:     ) -> dict[str, Any]:
7415:         """
```

```
        Execute complete analysis pipeline on a policy document file.

        Args:
            input_path: Path to input policy document (text format)
            output_path: Optional path for JSON results export

        Returns:
            Complete analysis results dictionary
        """
        input_path = Path(input_path)
        logger.info(f"Starting analysis of {input_path}")

        # Stage 1: Load document
        raw_text = ""
        suffix = input_path.suffix.lower()
        if suffix == ".pdf":
            raw_text = self.document_loader.load_pdf(str(input_path))
        elif suffix in {".docx", ".doc"}:
            raw_text = self.document_loader.load_docx(str(input_path))

        if not raw_text:
            raw_text = self.file_handler.read_text(input_path)
        logger.info(f"Loaded {len(raw_text)} characters from {input_path.name}")

        # Stage 2: Sanitize
        sanitized_text = self.sanitizer.sanitize(raw_text)
        reduction_pct = 100 * (1 - len(sanitized_text) / max(1, len(raw_text)))
        logger.info(f"Sanitization: {reduction_pct:.1f}% size reduction")

        # Stage 3: Process
        results = self.processor.process(sanitized_text)
        results["pipeline_metadata"] = {
            "input_file": str(input_path),
            "raw_size": len(raw_text),
            "sanitized_size": len(sanitized_text),
            "reduction_percentage": round(reduction_pct, 2),
        }

        # Stage 4: Export if requested
        if output_path:
            self.processor.export_results(results, output_path)

        logger.info(f"Analysis complete: {results['processing_status']}")
        return results

    @calibrated_method("farfan_core.processing.policy_processor.PolicyAnalysisPipeline.analyze_text")
    def analyze_text(self, raw_text: str) -> dict[str, Any]:
        """
        Execute analysis pipeline on raw text input.

        Args:
            raw_text: Raw policy document text

        Returns:
            Complete analysis results dictionary
        """
```

```
7472:         sanitized_text = self.sanitizer.sanitize(raw_text)
7473:         return self.processor.process(sanitized_text)
7474:
7475: # ==============================================================================
7476: # FACTORY FUNCTIONS FOR BACKWARD COMPATIBILITY
7477: # ==============================================================================
7478:
7479: def create_policy_processor(
7480:     preserve_structure: bool = True,
7481:     enable_semantic_tagging: bool = True,
7482:     confidence_threshold: float = ParameterLoaderV2.get("farfan_core.processing.policy_processor.PolicyAnalysisPipeline.analyze_text", "auto_param_L1595_34"
, 0.65),
7483:     **kwargs: Any,
7484: ) -> PolicyAnalysisPipeline:
7485:     """
7486:     Factory function for creating policy analysis pipeline with legacy support.
7487:
7488:     Args:
7489:         preserve_structure: Enable document structure preservation
7490:         enable_semantic_tagging: Enable semantic element tagging
7491:         confidence_threshold: Minimum confidence threshold for evidence
7492:         **kwargs: Additional configuration parameters
7493:
7494:     Returns:
7495:         Configured PolicyAnalysisPipeline instance
7496:     """
7497:     config = ProcessorConfig(
7498:         preserve_document_structure=preserve_structure,
7499:         enable_semantic_tagging=enable_semantic_tagging,
7500:         confidence_threshold=confidence_threshold,
7501:         **kwargs,
7502:     )
7503:     return PolicyAnalysisPipeline(config=config)
7504:
7505: # ==============================================================================
7506: # COMMAND-LINE INTERFACE
7507: # ==============================================================================
7508:
7509: def main() -> None:
7510:     """Command-line interface for policy plan analysis."""
7511:     import argparse
7512:
7513:     parser = argparse.ArgumentParser(
7514:         description="Industrial-Grade Policy Plan Processor for Colombian Local Development Plans"
7515:     )
7516:     parser.add_argument("input_file", type=str, help="Input policy document path")
7517:     parser.add_argument(
7518:         "-o", "--output", type=str, help="Output JSON file path", default=None
7519:     )
7520:     parser.add_argument(
7521:         "-t",
7522:         "--threshold",
7523:         type=float,
7524:         default=ParameterLoaderV2.get("farfan_core.processing.policy_processor.PolicyAnalysisPipeline.analyze_text", "auto_param_L1637_16", 0.65),
7525:         help="Confidence threshold (0-1)",
7526:     )
```

```
    parser.add_argument(
        "-q",
        "--questionnaire",
        type=str,
        help="Custom questionnaire JSON path",
        default=None,
    )
    parser.add_argument(
        "-v", "--verbose", action="store_true", help="Enable verbose logging"
    )

    args = parser.parse_args()

    if args.verbose:
        logging.getLogger().setLevel(logging.DEBUG)

    # Configure and execute pipeline
    config = ProcessorConfig(confidence_threshold=args.threshold)
    questionnaire_path = Path(args.questionnaire) if args.questionnaire else None

    pipeline = PolicyAnalysisPipeline(
        config=config, questionnaire_path=questionnaire_path
    )

    try:
        results = pipeline.analyze_file(args.input_file, args.output)

        # Print summary
        print("\n" + "=" * 70)
        print("POLICY ANALYSIS SUMMARY")
        print("=" * 70)
        print(f"Document: {results['metadata'].get('title', 'N/A')}")
        print(f"Entity: {results['metadata'].get('entity', 'N/A')}")
        print(f"Period: {results['metadata'].get('period', {})}")
        print(f"\nPolicy Points Covered: {results['document_statistics']['point_coverage']}")
        print(f"Average Confidence: {results['document_statistics']['avg_confidence']:.2%}")
        print(f"Total Sentences: {results['document_statistics']['sentence_count']}")
        print("=" * 70 + "\n")

    except Exception as e:
        logger.error(f"Analysis failed: {e}", exc_info=True)
        raise


================================================================================
FILE: src/farfan_pipeline/processing/quality_gates.py
================================================================================

"""
Quality gates for SPC (Smart Policy Chunks) validation - Canonical Phase-One.

Validates quality metrics, enforces invariants, and ensures compatibility
with downstream phases in the canonical pipeline flux.

MAXIMUM STANDARD: No tolerance for data quality degradation.
```

```
7583: """
7584:
7585: import logging
7586: from pathlib import Path
7587: from typing import Any
7588: # from farfan_core import get_parameter_loader  # CALIBRATION DISABLED
7589: from farfan_pipeline.core.calibration.decorators import calibrated_method
7590:
7591: logger = logging.getLogger(__name__)
7592:
7593:
7594: class SPCQualityGates:
7595:     """Quality validation gates for Smart Policy Chunks ingestion."""
7596:
7597:     # Phase-one output quality thresholds
7598:     MIN_CHUNKS = 5
7599:     MAX_CHUNKS = 200
7600:     MIN_CHUNK_LENGTH = 50  # characters
7601:     MAX_CHUNK_LENGTH = 5000
7602:     MIN_STRATEGIC_SCORE = 0.3
7603:     MIN_QUALITY_SCORE = 0.5
7604:     REQUIRED_CHUNK_FIELDS = ['text', 'chunk_id', 'strategic_importance', 'quality_score']
7605:
7606:     # Compatibility thresholds for downstream phases
7607:     MIN_EMBEDDING_DIM = 384  # For semantic analysis
7608:     REQUIRED_METADATA_FIELDS = ['document_id', 'title', 'version']
7609:
7610:     # CRITICAL quality metrics (per README specifications)
7611:     MIN_PROVENANCE_COMPLETENESS = 1.0  # 100% REQUIRED (no partial coverage tolerated)
7612:     MIN_STRUCTURAL_CONSISTENCY = 1.0   # 100% REQUIRED (perfect structure)
7613:     MIN_BOUNDARY_F1 = 0.85             # Chunk boundary quality
7614:     MIN_BUDGET_CONSISTENCY = 0.95      # Budget data consistency
7615:     MIN_TEMPORAL_ROBUSTNESS = 0.80     # Temporal data quality
7616:
7617:     @calibrated_method("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_input")
7618:     def validate_input(self, document_path: Path) -> dict[str, Any]:
7619:         """
7620:         Validate input document before processing.
7621:
7622:         Args:
7623:             document_path: Path to input document
7624:
7625:         Returns:
7626:             Dictionary with validation results
7627:         """
7628:         failures = []
7629:
7630:         # Check file exists
7631:         if not document_path.exists():
7632:             failures.append(f"Input document not found: {document_path}")
7633:             return {"passed": False, "failures": failures}
7634:
7635:         # Check file size (not empty, not too large)
7636:         file_size = document_path.stat().st_size
7637:         if file_size == 0:
7638:             failures.append("Input document is empty")
```

```
        elif file_size > 50 * 1024 * 1024:  # 50MB limit
            failures.append(f"Input document too large: {file_size / 1024 / 1024:.1f}MB")

        # Check file extension
        if document_path.suffix.lower() not in ['.txt', '.pdf', '.json']:
            failures.append(f"Unsupported file type: {document_path.suffix}")

        return {
            "passed": len(failures) == 0,
            "failures": failures,
            "file_size_bytes": file_size,
        }

    @calibrated_method("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_chunks")
    def validate_chunks(self, chunks: list[dict[str, Any]]) -> dict[str, Any]:
        """
        Validate processed chunks from phase-one.

        Args:
            chunks: List of smart policy chunks

        Returns:
            Dictionary with validation results
        """
        failures = []
        warnings = []

        # Check chunk count
        if len(chunks) < self.MIN_CHUNKS:
            failures.append(f"Too few chunks: {len(chunks)} < {self.MIN_CHUNKS}")
        elif len(chunks) > self.MAX_CHUNKS:
            warnings.append(f"High chunk count: {len(chunks)} > {self.MAX_CHUNKS}")

        # Validate each chunk
        for idx, chunk in enumerate(chunks):
            chunk_id = chunk.get('chunk_id', f'chunk_{idx}')

            # Check required fields
            for field in self.REQUIRED_CHUNK_FIELDS:
                if field not in chunk:
                    failures.append(f"{chunk_id}: Missing required field '{field}'")

            # Check chunk text length
            text = chunk.get('text', '')
            if len(text) < self.MIN_CHUNK_LENGTH:
                failures.append(f"{chunk_id}: Text too short: {len(text)} < {self.MIN_CHUNK_LENGTH}")
            elif len(text) > self.MAX_CHUNK_LENGTH:
                warnings.append(f"{chunk_id}: Text very long: {len(text)} > {self.MAX_CHUNK_LENGTH}")

            # Check quality scores
            strategic_score = chunk.get('strategic_importance', 0)
            if strategic_score < self.MIN_STRATEGIC_SCORE:
                warnings.append(f"{chunk_id}: Low strategic importance: {strategic_score}")

            quality_score = chunk.get('quality_score', 0)
            if quality_score < self.MIN_QUALITY_SCORE:
```

```
                warnings.append(f"{chunk_id}: Low quality score: {quality_score}")

        return {
            "passed": len(failures) == 0,
            "failures": failures,
            "warnings": warnings,
            "chunk_count": len(chunks),
        }

    @calibrated_method("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_output_compatibility")
    def validate_output_compatibility(self, output: dict[str, Any]) -> dict[str, Any]:
        """
        Validate output structure for compatibility with downstream phases.

        Ensures the phase-one output can be consumed by next phases in the flux.

        Args:
            output: Phase-one output dictionary

        Returns:
            Dictionary with validation results
        """
        failures = []

        # Check required top-level keys
        if 'chunks' not in output:
            failures.append("Missing 'chunks' in output")

        if 'metadata' not in output:
            failures.append("Missing 'metadata' in output")
        else:
            # Validate metadata fields
            metadata = output['metadata']
            for field in self.REQUIRED_METADATA_FIELDS:
                if field not in metadata:
                    failures.append(f"Missing required metadata field: '{field}'")

        # Check chunks structure
        if 'chunks' in output:
            chunks_result = self.validate_chunks(output['chunks'])
            if not chunks_result['passed']:
                failures.extend(chunks_result['failures'])

        return {
            "passed": len(failures) == 0,
            "failures": failures,
        }

    @calibrated_method("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics")
    def validate_quality_metrics(self, quality_metrics: Any) -> dict[str, Any]:
        """
        Validate quality metrics from CanonPolicyPackage against MAXIMUM STANDARDS.

        Enforces strict thresholds per README specifications. No degradation tolerated.

        Args:
```

```
            quality_metrics: QualityMetrics instance from CanonPolicyPackage

        Returns:
            Dictionary with validation results:
            {
                "passed": bool,
                "failures": List[str],  # CRITICAL failures that MUST be fixed
                "warnings": List[str],  # Non-critical warnings
                "metrics": Dict[str, float]  # Actual metric values
            }
        """
        failures = []
        warnings = []
        metrics_dict = {}

        # Extract metrics (handle both object and dict)
        if hasattr(quality_metrics, '__dict__'):
            # It's an object
            provenance_completeness = getattr(quality_metrics, 'provenance_completeness', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L193_90", 0.0))
            structural_consistency = getattr(quality_metrics, 'structural_consistency', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L194_88", 0.0))
            boundary_f1 = getattr(quality_metrics, 'boundary_f1', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L195_66", 0.0))
            budget_consistency = getattr(quality_metrics, 'budget_consistency_score', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L196_86", 0.0))
            temporal_robustness = getattr(quality_metrics, 'temporal_robustness', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L197_82", 0.0))
            chunk_context_coverage = getattr(quality_metrics, 'chunk_context_coverage', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L198_88", 0.0))
        else:
            # It's a dict
            provenance_completeness = quality_metrics.get('provenance_completeness', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L201_85", 0.0))
            structural_consistency = quality_metrics.get('structural_consistency', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L202_83", 0.0))
            boundary_f1 = quality_metrics.get('boundary_f1', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L203_61", 0.0))
            budget_consistency = quality_metrics.get('budget_consistency_score', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L204_81", 0.0))
            temporal_robustness = quality_metrics.get('temporal_robustness', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L205_77", 0.0))
            chunk_context_coverage = quality_metrics.get('chunk_context_coverage', ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "auto_param_L206_83", 0.0))

        # Store actual values
        metrics_dict = {
            'provenance_completeness': provenance_completeness,
            'structural_consistency': structural_consistency,
            'boundary_f1': boundary_f1,
            'budget_consistency_score': budget_consistency,
            'temporal_robustness': temporal_robustness,
            'chunk_context_coverage': chunk_context_coverage,
        }

        # CRITICAL: Provenance completeness MUST be 100%
```

```
        if provenance_completeness < self.MIN_PROVENANCE_COMPLETENESS:
            failures.append(
                f"ð\237\224´ CRITICAL: Provenance completeness below threshold: "
                f"{provenance_completeness:.2%} < {self.MIN_PROVENANCE_COMPLETENESS:.0%}. "
                f"Every token must be traceable to source (README requirement)."
            )
            logger.error(
                f"Provenance completeness FAILED: {provenance_completeness:.2%} "
                f"(required: {self.MIN_PROVENANCE_COMPLETENESS:.0%})"
            )

        # CRITICAL: Structural consistency MUST be perfect
        if structural_consistency < self.MIN_STRUCTURAL_CONSISTENCY:
            failures.append(
                f"ð\237\224´ CRITICAL: Structural consistency below threshold: "
                f"{structural_consistency:.2%} < {self.MIN_STRUCTURAL_CONSISTENCY:.0%}. "
                f"Policy structure must be perfectly parsed (FASE 3 gate)."
            )
            logger.error(
                f"Structural consistency FAILED: {structural_consistency:.2%} "
                f"(required: {self.MIN_STRUCTURAL_CONSISTENCY:.0%})"
            )

        # HIGH: Boundary F1 for chunk quality
        if boundary_f1 < self.MIN_BOUNDARY_F1:
            failures.append(
                f"ð\237\224´ HIGH: Boundary F1 below threshold: "
                f"{boundary_f1:.2f} < {self.MIN_BOUNDARY_F1}. "
                f"Chunk boundaries are not accurate enough (FASE 8 gate)."
            )
            logger.error(
                f"Boundary F1 FAILED: {boundary_f1:.2f} "
                f"(required: {self.MIN_BOUNDARY_F1})"
            )

        # HIGH: Budget consistency for financial data
        if budget_consistency < self.MIN_BUDGET_CONSISTENCY:
            warnings.append(
                f"ð\237\237¡ Budget consistency below threshold: "
                f"{budget_consistency:.2%} < {self.MIN_BUDGET_CONSISTENCY:.0%}. "
                f"Budget data may have inconsistencies (FASE 6 gate)."
            )
            logger.warning(
                f"Budget consistency WARNING: {budget_consistency:.2%} "
                f"(recommended: {self.MIN_BUDGET_CONSISTENCY:.0%})"
            )

        # MEDIUM: Temporal robustness
        if temporal_robustness < self.MIN_TEMPORAL_ROBUSTNESS:
            warnings.append(
                f"ð\237\237¡ Temporal robustness below threshold: "
                f"{temporal_robustness:.2%} < {self.MIN_TEMPORAL_ROBUSTNESS:.0%}. "
                f"Temporal data may be incomplete."
            )

        # INFO: Chunk context coverage
```

```
 7851:         if chunk_context_coverage < ParameterLoaderV2.get("farfan_core.processing.spc_ingestion.quality_gates.SPCQualityGates.validate_quality_metrics", "au
to_param_L275_36", 0.5):
 7852:             warnings.append(
 7853:                 f"â\204¹ï¸\217 Low chunk context coverage: {chunk_context_coverage:.2%}. "
 7854:                 f"Few inter-chunk relationships detected."
 7855:             )
 7856:
 7857:         # Summary logging
 7858:         if failures:
 7859:             logger.error(f"Quality metrics validation FAILED with {len(failures)} critical issues")
 7860:         elif warnings:
 7861:             logger.warning(f"Quality metrics validation PASSED with {len(warnings)} warnings")
 7862:         else:
 7863:             logger.info("Quality metrics validation PASSED - All thresholds met")
 7864:
 7865:         return {
 7866:             "passed": len(failures) == 0,
 7867:             "failures": failures,
 7868:             "warnings": warnings,
 7869:             "metrics": metrics_dict,
 7870:         }
 7871:
 7872:
 7873: # Legacy alias for backwards compatibility
 7874: class QualityGates(SPCQualityGates):
 7875:     """Legacy alias for SPCQualityGates."""
 7876:     pass
 7877:
 7878:
 7879:
 7880: ================================================================================
 7881: FILE: src/farfan_pipeline/processing/semantic_chunking_policy.py
 7882: ================================================================================
 7883:
 7884: """
 7885: INTERNAL SPC COMPONENT
 7886:
 7887: â\232 ï¸\217  USAGE RESTRICTION â\232 ï¸\217
 7888: ================================================================================
 7889: This module implements SOTA semantic chunking and policy analysis for Smart
 7890: Policy Chunks. It MUST NOT be used as a standalone ingestion pipeline in the
 7891: canonical FARFAN flow.
 7892:
 7893: Canonical entrypoint is scripts/run_policy_pipeline_verified.py.
 7894:
 7895: This module is an INTERNAL COMPONENT of:
 7896:     src/farfan_core/processing/spc_ingestion.py (StrategicChunkingSystem)
 7897:
 7898: DO NOT use this module directly as an independent pipeline. It is consumed
 7899: internally by the SPC core and should only be imported from within:
 7900:     - farfan_core.processing.spc_ingestion
 7901:     - Unit tests for SPC components
 7902:
 7903: Scientific Foundation:
 7904: - Semantic: BGE-M3 (2024, SOTA multilingual dense retrieval)
 7905: - Chunking: Semantic-aware with policy structure recognition
```

```
- Math: Information-theoretic Bayesian evidence accumulation
- Causal: Directed Acyclic Graph inference with interventional calculus
================================================================================
"""
from __future__ import annotations

import json
import logging
import re
from dataclasses import dataclass
from enum import Enum
from typing import TYPE_CHECKING, Any, Literal

import numpy as np
import torch
from scipy import stats
from scipy.spatial.distance import cosine
from scipy.special import rel_entr

# Check dependency lockdown before importing transformers
from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
from transformers import AutoModel, AutoTokenizer
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

_lockdown = get_dependency_lockdown()

if TYPE_CHECKING:
    from numpy.typing import NDArray

# Note: logging.basicConfig should be called by the application entry point,
# not at module import time to avoid side effects
logger = logging.getLogger("policy_framework")

def _get_chunk_content(chunk: dict[str, Any]) -> str:
    """Compatibility helper returning the canonical chunk content field."""

    if "content" in chunk:
        return chunk["content"]
    return chunk.get("text", "")

def _upgrade_chunk_schema(chunk: dict[str, Any]) -> dict[str, Any]:
    """Return a chunk dict that guarantees ``content`` availability."""

    if "content" in chunk:
        return chunk
    upgraded = dict(chunk)
    upgraded["content"] = upgraded.get("text", "")
    return upgraded

# ========================
# CALIBRATED CONSTANTS (SOTA)
# ========================
POSITION_WEIGHT_SCALE: float = 0.42  # Early sections exert stronger evidentiary leverage
TABLE_WEIGHT_FACTOR: float = 1.35  # Tabular content is typically audited data
NUMERICAL_WEIGHT_FACTOR: float = 1.18  # Numerical narratives reinforce credibility
```

```
7962: PLAN_SECTION_WEIGHT_FACTOR: float = 1.25  # Investment plans anchor execution feasibility
7963: DIAGNOSTIC_SECTION_WEIGHT_FACTOR: float = 0.92  # Diagnostics contextualize but do not commit resources
7964: RENYI_ALPHA_ORDER: float = 1.45  # Van Erven & HarremoÃ«s (2014) Optimum between KL and RÃ©nyi regimes
7965: RENYI_ALERT_THRESHOLD: float = 0.24  # Empirically tuned on 2021-2024 Colombian PDM corpus
7966: RENYI_CURVATURE_GAIN: float = 0.85  # Amplifies curvature impact without destabilizing evidence
7967: RENYI_FLUX_TEMPERATURE: float = 0.65  # Controls saturation of Renyi coherence flux
7968: RENYI_STABILITY_EPSILON: float = 1e-9  # Numerical guard-rail for degenerative posteriors
7969:
7970: # ========================
7971: # DOMAIN ONTOLOGY
7972: # ========================
7973:
7974: class CausalDimension(Enum):
7975:     """Marco LÃ³gico standard (DNP Colombia)"""
7976:     INSUMOS = "insumos"  # Recursos, capacidad institucional
7977:     ACTIVIDADES = "actividades"  # Acciones, procesos, cronogramas
7978:     PRODUCTOS = "productos"  # Entregables inmediatos
7979:     RESULTADOS = "resultados"  # Efectos mediano plazo
7980:     IMPACTOS = "impactos"  # TransformaciÃ³n estructural largo plazo
7981:     SUPUESTOS = "supuestos"  # Condiciones habilitantes
7982:
7983: class PDMSection(Enum):
7984:     """
7985:     Enumerates the typical sections of a Colombian Municipal Development Plan (PDM),
7986:     as defined by Ley 152/1994. Each member represents a key structural component
7987:     of the PDM document, facilitating semantic analysis and policy structure recognition.
7988:     """
7989:     DIAGNOSTICO = "diagnostico"
7990:     VISION_ESTRATEGICA = "vision_estrategica"
7991:     PLAN_PLURIANUAL = "plan_plurianual"
7992:     PLAN_INVERSIONES = "plan_inversiones"
7993:     MARCO_FISCAL = "marco_fiscal"
7994:     SEGUIMIENTO = "seguimiento_evaluacion"
7995:
7996: @dataclass(frozen=True, slots=True)
7997: class SemanticConfig:
7998:     """ConfiguraciÃ³n calibrada para anÃ¡lisis de polÃticas pÃºblicas"""
7999:     # BGE-M3: Best multilingual embedding (Jan 2024, beats E5)
8000:     embedding_model: str = "BAAI/bge-m3"
8001:     chunk_size: int = 768  # Optimal for policy paragraphs (empirical)
8002:     chunk_overlap: int = 128  # Preserve cross-boundary context
8003:     similarity_threshold: float = 0.82  # Calibrated on PDM corpus
8004:     min_evidence_chunks: int = 3  # Statistical significance floor
8005:     bayesian_prior_strength: float = 0.5  # Conservative uncertainty
8006:     device: Literal["cpu", "cuda"] | None = None
8007:     batch_size: int = 32
8008:     fp16: bool = True  # Memory optimization
8009:
8010: # ========================
8011: # SEMANTIC PROCESSOR (SOTA)
8012: # ========================
8013:
8014: class SemanticProcessor:
8015:     """
8016:     State-of-the-art semantic processing with:
8017:     - BGE-M3 embeddings (2024 SOTA)
```

```
    - Policy-aware chunking (respects PDM structure)
    - Efficient batching with FP16
    """

    def __init__(self, config: SemanticConfig) -> None:
        self.config = config
        self._model = None
        self._tokenizer = None
        self._loaded = False

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor._lazy_load")
    def _lazy_load(self) -> None:
        if self._loaded:
            return
        try:
            device = self.config.device or ("cuda" if torch.cuda.is_available() else "cpu")
            logger.info(f"Loading BGE-M3 model on {device}...")
            self._tokenizer = AutoTokenizer.from_pretrained(self.config.embedding_model)
            self._model = AutoModel.from_pretrained(
                self.config.embedding_model,
                torch_dtype=torch.float16 if self.config.fp16 and device == "cuda" else torch.float32
            ).to(device)
            self._model.eval()
            self._loaded = True
            logger.info("BGE-M3 loaded successfully")
        except ImportError as e:
            missing = None
            msg = str(e)
            if "transformers" in msg:
                missing = "transformers"
            elif "torch" in msg:
                missing = "torch"
            else:
                missing = "transformers or torch"
            raise RuntimeError(
                f"Missing dependency: {missing}. Please install with 'pip install {missing}'"
            ) from e

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor.chunk_text")
    def chunk_text(self, text: str, preserve_structure: bool = True) -> list[dict[str, Any]]:
        """
        Policy-aware semantic chunking:
        - Respects section boundaries (numbered lists, headers)
        - Maintains table integrity
        - Preserves reference links between text segments
        """
        self._lazy_load()
        # Detect structural elements (headings, numbered sections, tables)
        if preserve_structure:
            sections = self._detect_pdm_structure(text)
        else:
            sections = [{"text": text, "type": "TEXT", "id": 0}]
        chunks = []
        for section in sections:
            # Tokenize section
            tokens = self._tokenizer.encode(
```

```
                section["text"],
                add_special_tokens=False,
                truncation=False
            )
            # Sliding window with overlap
            for i in range(0, len(tokens), self.config.chunk_size - self.config.chunk_overlap):
                chunk_tokens = tokens[i:i + self.config.chunk_size]
                chunk_text = self._tokenizer.decode(chunk_tokens, skip_special_tokens=True)
                chunks.append({
                    "content": chunk_text,
                    "section_type": section["type"],
                    "section_id": section["id"],
                    "token_count": len(chunk_tokens),
                    "position": len(chunks),
                    "has_table": self._detect_table(chunk_text),
                    "has_numerical": self._detect_numerical_data(chunk_text),
                    "pdq_context": {},
                })
        # Batch embed all chunks
        embeddings = self._embed_batch([c["content"] for c in chunks])
        for chunk, emb in zip(chunks, embeddings, strict=False):
            chunk["embedding"] = emb
        logger.info(f"Generated {len(chunks)} policy-aware chunks")
        return [_upgrade_chunk_schema(chunk) for chunk in chunks]

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor._detect_pdm_structure")
    def _detect_pdm_structure(self, text: str) -> list[dict[str, Any]]:
        """Detect PDM sections using Colombian policy document patterns"""
        sections = []
        # Patterns for Colombian PDM structure
        patterns = {
            PDMSection.DIAGNOSTICO: r"(?i)(diagnÃ³stico|caracterizaciÃ³n|situaciÃ³n actual)",
            PDMSection.VISION_ESTRATEGICA: r"(?i)(visiÃ³n|misiÃ³n|objetivos estratÃ©gicos)",
            PDMSection.PLAN_PLURIANUAL: r"(?i)(plan plurianual|programas|proyectos)",
            PDMSection.PLAN_INVERSIONES: r"(?i)(plan de inversiones|presupuesto|recursos)",
            PDMSection.MARCO_FISCAL: r"(?i)(marco fiscal|sostenibilidad fiscal)",
            PDMSection.SEGUIMIENTO: r"(?i)(seguimiento|evaluaciÃ³n|indicadores)"
        }
        # Split by major headers (numbered or capitalized)
        parts = re.split(r'\n(?=[0-9]+\.|[A-ZÃ\221Ã\201Ã\211Ã\215Ã\223Ã\232]{3,})', text)
        for i, part in enumerate(parts):
            section_type = PDMSection.DIAGNOSTICO  # default
            for stype, pattern in patterns.items():
                if re.search(pattern, part[:200]):
                    section_type = stype
                    break
            sections.append({
                "text": part.strip(),
                "type": section_type,
                "id": f"sec_{i}"
            })
        return sections

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor._detect_table")
    def _detect_table(self, text: str) -> bool:
        """Detect if chunk contains tabular data"""
```

```
8130:         # Multiple tabs or pipes suggest table structure
8131:         return (text.count('\t') > 3 or
8132:                 text.count('|') > 3 or
8133:                 bool(re.search(r'\d+\s+\d+\s+\d+', text)))
8134:
8135:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor._detect_numerical_data")
8136:     def _detect_numerical_data(self, text: str) -> bool:
8137:         """Detect if chunk contains significant numerical/financial data"""
8138:         # Look for currency, percentages, large numbers
8139:         patterns = [
8140:             r'\$\s*\d+(?:[\.,]\d+)*',  # Currency
8141:             r'\d+(?:[\.,]\d+)*\s*%',  # Percentages
8142:             r'\d{1,3}(?:[.,]\d{3})+',  # Large numbers with separators
8143:         ]
8144:         return any(re.search(p, text) for p in patterns)
8145:
8146:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor._embed_batch")
8147:     def _embed_batch(self, texts: list[str]) -> list[NDArray[np.floating[Any]]]:
8148:         """Batch embedding with BGE-M3"""
8149:         self._lazy_load()
8150:         embeddings = []
8151:         for i in range(0, len(texts), self.config.batch_size):
8152:             batch = texts[i:i + self.config.batch_size]
8153:             # Tokenize batch
8154:             encoded = self._tokenizer(
8155:                 batch,
8156:                 padding=True,
8157:                 truncation=True,
8158:                 max_length=self.config.chunk_size,
8159:                 return_tensors="pt"
8160:             ).to(self._model.device)
8161:             # Generate embeddings (mean pooling)
8162:             with torch.no_grad():
8163:                 outputs = self._model(**encoded)
8164:             # Mean pooling over sequence
8165:             attention_mask = encoded["attention_mask"]
8166:             token_embeddings = outputs.last_hidden_state
8167:             input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
8168:             sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
8169:             sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
8170:             batch_embeddings = (sum_embeddings / sum_mask).cpu().numpy()
8171:             embeddings.extend([emb.astype(np.float32) for emb in batch_embeddings])
8172:         return embeddings
8173:
8174:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticProcessor.embed_single")
8175:     def embed_single(self, text: str) -> NDArray[np.floating[Any]]:
8176:         """Single text embedding"""
8177:         return self._embed_batch([text])[0]
8178:
8179: # ========================
8180: # MATHEMATICAL ENHANCER (RIGOROUS)
8181: # ========================
8182:
8183: class BayesianEvidenceIntegrator:
8184:     """
8185:     Information-theoretic Bayesian evidence accumulation:
```

```
8186:     - Dirichlet-Multinomial for multi-hypothesis tracking
8187:     - KL divergence for belief update quantification
8188:     - Entropy-based confidence calibration
8189:     - No simplifications or heuristics
8190:     """
8191:
8192:     def __init__(self, prior_concentration: float = 0.5) -> None:
8193:         """
8194:         Args:
8195:             prior_concentration: Dirichlet concentration (Î±).
8196:                 Lower = more uncertain prior (conservative)
8197:         """
8198:         if prior_concentration <= 0:
8199:             raise ValueError(
8200:                 f"Invalid prior_concentration: Dirichlet concentration parameter must be strictly positive. "
8201:                 f"Received: {prior_concentration}"
8202:             )
8203:         self.prior_alpha = float(prior_concentration)
8204:
8205:     def integrate_evidence(
8206:         self,
8207:         similarities: NDArray[np.float64],
8208:         chunk_metadata: list[dict[str, Any]]
8209:     ) -> dict[str, float]:
8210:         """
8211:         Bayesian evidence integration with information-theoretic rigor:
8212:         1. Map similarities to likelihood space via monotonic transform
8213:         2. Weight evidence by chunk reliability (position, structure, content type)
8214:         3. Update Dirichlet posterior
8215:         4. Compute information gain (KL divergence from prior)
8216:         5. Calculate calibrated confidence with epistemic uncertainty
8217:         """
8218:         if len(similarities) == 0:
8219:             return self._null_evidence()
8220:         # 1. Transform similarities to probability space
8221:         # Using sigmoid with learned temperature for calibration
8222:         sims = np.asarray(similarities, dtype=np.float64)
8223:         probs = self._similarity_to_probability(sims)
8224:         # 2. Compute reliability weights from metadata
8225:         weights = self._compute_reliability_weights(chunk_metadata)
8226:         # 3. Aggregate weighted evidence
8227:         # Dirichlet posterior parameters: Î±_post = Î±_prior + weighted_counts
8228:         positive_evidence = np.sum(weights * probs)
8229:         negative_evidence = np.sum(weights * (ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__", "
auto_param_L348_46", 1.0) - probs))
8230:         alpha_pos = self.prior_alpha + positive_evidence
8231:         alpha_neg = self.prior_alpha + negative_evidence
8232:         alpha_total = alpha_pos + alpha_neg
8233:         # 4. Posterior statistics
8234:         posterior_mean = alpha_pos / alpha_total
8235:         posterior_variance = (alpha_pos * alpha_neg) / (
8236:             alpha_total**2 * (alpha_total + 1)
8237:         )
8238:         # 5. Information gain (KL divergence from prior to posterior)
8239:         prior_dist = np.array([self.prior_alpha, self.prior_alpha])
8240:         prior_dist = prior_dist / prior_dist.sum()
```

```
 8241:         posterior_dist = np.array([alpha_pos, alpha_neg])
 8242:         posterior_dist = posterior_dist / posterior_dist.sum()
 8243:         kl_divergence = float(np.sum(rel_entr(posterior_dist, prior_dist)))
 8244:         # 6. Entropy-based calibrated confidence
 8245:         posterior_entropy = stats.beta.entropy(alpha_pos, alpha_neg)
 8246:         max_entropy = stats.beta.entropy(1, 1)  # Maximum uncertainty
 8247:         confidence = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__", "auto_param_L366_21", 1.0)
 - (posterior_entropy / max_entropy)
 8248:         return {
 8249:             "posterior_mean": float(np.clip(posterior_mean, ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrato
r.__init__", "auto_param_L368_60", 0.0), ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__", "auto_param_L
368_65", 1.0))),
 8250:             "posterior_std": float(np.sqrt(posterior_variance)),
 8251:             "information_gain": float(kl_divergence),
 8252:             "confidence": float(confidence),
 8253:             "evidence_strength": float(
 8254:                 positive_evidence / (alpha_total - 2 * self.prior_alpha)
 8255:                 if abs(alpha_total - 2 * self.prior_alpha) > 1e-8 else ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEviden
ceIntegrator.__init__", "auto_param_L374_71", 0.0)
 8256:             ),
 8257:             "n_chunks": len(similarities)
 8258:         }
 8259:
 8260:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._similarity_to_probability")
 8261:     def _similarity_to_probability(self, sims: NDArray[np.float64]) -> NDArray[np.float64]:
 8262:         """
 8263:         Calibrated transform from cosine similarity [-1,1] to probability [0,1]
 8264:         Using sigmoid with empirically derived temperature
 8265:         """
 8266:         # Shift to [0,2], scale to reasonable range
 8267:         x = (sims + ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._similarity_to_probability", "auto_par
am_L386_20", 1.0)) * 2.0
 8268:         # Sigmoid with temperature=2.0 (calibrated on policy corpus)
 8269:         return ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._similarity_to_probability", "auto_param_L3
88_15", 1.0) / (ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._similarity_to_probability", "auto_param_L388_22"
, 1.0) + np.exp(-x / 2.0))
 8270:
 8271:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._compute_reliability_weights")
 8272:     def _compute_reliability_weights(self, metadata: list[dict[str, Any]]) -> NDArray[np.float64]:
 8273:         """
 8274:         Evidence reliability based on:
 8275:         - Position in document (early sections more diagnostic)
 8276:         - Content type (tables/numbers more reliable for quantitative claims)
 8277:         - Section type (plan sections more reliable than diagnostics)
 8278:         """
 8279:         n = len(metadata)
 8280:         weights = np.ones(n, dtype=np.float64)
 8281:         for i, meta in enumerate(metadata):
 8282:             # Position weight (early = more reliable)
 8283:             pos_weight = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._compute_reliability_weights", "a
uto_param_L402_25", 1.0) - (meta["position"] / max(1, n)) * POSITION_WEIGHT_SCALE
 8284:             # Content type weight
 8285:             content_weight = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._compute_reliability_weights"
, "content_weight", 1.0) # Refactored
 8286:             if meta.get("has_table", False):
 8287:                 content_weight *= TABLE_WEIGHT_FACTOR
```

```
 8288:             if meta.get("has_numerical", False):
 8289:                 content_weight *= NUMERICAL_WEIGHT_FACTOR
 8290:             # Section type weight (plan sections > diagnostic)
 8291:             section_type = meta.get("section_type")
 8292:             if section_type in [PDMSection.PLAN_PLURIANUAL, PDMSection.PLAN_INVERSIONES]:
 8293:                 content_weight *= PLAN_SECTION_WEIGHT_FACTOR
 8294:             elif section_type == PDMSection.DIAGNOSTICO:
 8295:                 content_weight *= DIAGNOSTIC_SECTION_WEIGHT_FACTOR
 8296:             weights[i] = pos_weight * content_weight
 8297:         # Normalize to sum to n (preserve total evidence mass)
 8298:         return weights * (n / weights.sum())
 8299:
 8300:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence")
 8301:     def _null_evidence(self) -> dict[str, float]:
 8302:         """Return prior state (no evidence)"""
 8303:         prior_mean = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "prior_mean", 0.5) #
 Refactored
 8304:         prior_var = self.prior_alpha / \
 8305:             ((2 * self.prior_alpha)**2 * (2 * self.prior_alpha + 1))
 8306:         return {
 8307:             "posterior_mean": prior_mean,
 8308:             "posterior_std": float(np.sqrt(prior_var)),
 8309:             "information_gain": ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_par
am_L428_32", 0.0),
 8310:             "confidence": ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L42
9_26", 0.0),
 8311:             "evidence_strength": ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_pa
ram_L430_33", 0.0),
 8312:             "n_chunks": 0
 8313:         }
 8314:
 8315:     def causal_strength(
 8316:         self,
 8317:         cause_emb: NDArray[np.floating[Any]],
 8318:         effect_emb: NDArray[np.floating[Any]],
 8319:         context_emb: NDArray[np.floating[Any]]
 8320:     ) -> float:
 8321:         """
 8322:         Causal strength via conditional independence approximation:
 8323:         strength = sim(cause, effect) * [1 - |sim(cause,ctx) - sim(effect,ctx)|]
 8324:         Intuition: Strong causal link if cause-effect similar AND
 8325:         both relate similarly to context (conditional independence test proxy)
 8326:         """
 8327:         sim_ce = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L446_17", 1.
0) - cosine(cause_emb, effect_emb)
 8328:         sim_c_ctx = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L447_20",
 1.0) - cosine(cause_emb, context_emb)
 8329:         sim_e_ctx = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L448_20",
 1.0) - cosine(effect_emb, context_emb)
 8330:         # Conditional independence proxy
 8331:         cond_indep = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L450_21"
, 1.0) - abs(sim_c_ctx - sim_e_ctx)
 8332:         # Combined strength (normalized to [0,1])
 8333:         strength = ((sim_ce + 1) / 2) * cond_indep
 8334:         return float(np.clip(strength, ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "a
uto_param_L453_39", 0.0), ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.BayesianEvidenceIntegrator._null_evidence", "auto_param_L453_44",
```

```
1.0)))
 8335:
 8336: # ========================
 8337: # POLICY ANALYZER (INTEGRATED)
 8338: # ========================
 8339:
 8340: class PolicyDocumentAnalyzer:
 8341:     """
 8342:     Colombian Municipal Development Plan Analyzer:
 8343:     - BGE-M3 semantic processing
 8344:     - Policy-aware chunking (respects PDM structure)
 8345:     - Bayesian evidence integration with information theory
 8346:     - Causal dimension analysis per Marco LÃ³gico
 8347:     """
 8348:
 8349:     def __init__(self, config: SemanticConfig | None = None) -> None:
 8350:         self.config = config or SemanticConfig()
 8351:         self.semantic = SemanticProcessor(self.config)
 8352:         self.bayesian = BayesianEvidenceIntegrator(
 8353:             prior_concentration=self.config.bayesian_prior_strength
 8354:         )
 8355:         # Initialize dimension embeddings
 8356:         self.dimension_embeddings = self._init_dimension_embeddings()
 8357:
 8358:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.PolicyDocumentAnalyzer._init_dimension_embeddings")
 8359:     def _init_dimension_embeddings(self) -> dict[CausalDimension, NDArray[np.floating[Any]]]:
 8360:         """
 8361:         Canonical embeddings for Marco LÃ³gico dimensions
 8362:         Using Colombian policy-specific terminology
 8363:         """
 8364:         descriptions = {
 8365:             CausalDimension.INSUMOS: (
 8366:                 "recursos humanos financieros tÃ©cnicos capacidad institucional "
 8367:                 "presupuesto asignado infraestructura disponible personal capacitado"
 8368:             ),
 8369:             CausalDimension.ACTIVIDADES: (
 8370:                 "actividades programadas acciones ejecutadas procesos implementados "
 8371:                 "cronograma cumplido capacitaciones realizadas gestiones adelantadas"
 8372:             ),
 8373:             CausalDimension.PRODUCTOS: (
 8374:                 "productos entregables resultados inmediatos bienes servicios generados "
 8375:                 "documentos producidos obras construidas beneficiarios atendidos"
 8376:             ),
 8377:             CausalDimension.RESULTADOS: (
 8378:                 "resultados efectos mediano plazo cambios comportamiento acceso mejorado "
 8379:                 "capacidades fortalecidas servicios prestados metas alcanzadas"
 8380:             ),
 8381:             CausalDimension.IMPACTOS: (
 8382:                 "impactos transformaciÃ³n estructural efectos largo plazo desarrollo sostenible "
 8383:                 "bienestar poblacional reducciÃ³n pobreza equidad territorial"
 8384:             ),
 8385:             CausalDimension.SUPUESTOS: (
 8386:                 "supuestos condiciones habilitantes riesgos externos factores contextuales "
 8387:                 "viabilidad polÃtica sostenibilidad financiera apropiaciÃ³n comunitaria"
 8388:             )
 8389:         }
```

```
8390:         return {
8391:             dim: self.semantic.embed_single(desc)
8392:             for dim, desc in descriptions.items()
8393:         }
8394:
8395:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.PolicyDocumentAnalyzer.analyze")
8396:     def analyze(self, text: str) -> dict[str, Any]:
8397:         """
8398:         Full pipeline: chunking â\206\222 embedding â\206\222 dimension analysis â\206\222 evidence integration
8399:         """
8400:         # 1. Policy-aware chunking
8401:         chunks = self.semantic.chunk_text(text, preserve_structure=True)
8402:         logger.info(f"Processing {len(chunks)} chunks")
8403:         # 2. Analyze each causal dimension
8404:         dimension_results = {}
8405:         for dim, dim_emb in self.dimension_embeddings.items():
8406:             similarities = np.array([
8407:                 ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.PolicyDocumentAnalyzer.analyze", "auto_param_L526_16", 1.0) - cosine(
chunk["embedding"], dim_emb)
8408:                 for chunk in chunks
8409:             ])
8410:             # Filter by threshold
8411:             relevant_mask = similarities >= self.config.similarity_threshold
8412:             relevant_sims = similarities[relevant_mask]
8413:             relevant_chunks = [c for c, m in zip(chunks, relevant_mask, strict=False) if m]
8414:             # Bayesian integration
8415:             if len(relevant_sims) >= self.config.min_evidence_chunks:
8416:                 evidence = self.bayesian.integrate_evidence(
8417:                     relevant_sims,
8418:                     relevant_chunks
8419:                 )
8420:             else:
8421:                 evidence = self.bayesian._null_evidence()
8422:             dimension_results[dim.value] = {
8423:                 "total_chunks": int(np.sum(relevant_mask)),
8424:                 "mean_similarity": float(np.mean(similarities)),
8425:                 "max_similarity": float(np.max(similarities)),
8426:                 **evidence
8427:             }
8428:         # 3. Extract key findings (top chunks per dimension)
8429:         key_excerpts = self._extract_key_excerpts(chunks, dimension_results)
8430:         return {
8431:             "summary": {
8432:                 "total_chunks": len(chunks),
8433:                 "sections_detected": len({c["section_type"] for c in chunks}),
8434:                 "has_tables": sum(1 for c in chunks if c["has_table"]),
8435:                 "has_numerical": sum(1 for c in chunks if c["has_numerical"])
8436:             },
8437:             "causal_dimensions": dimension_results,
8438:             "key_excerpts": key_excerpts
8439:         }
8440:
8441:     def _extract_key_excerpts(
8442:         self,
8443:         chunks: list[dict[str, Any]],
8444:         dimension_results: dict[str, dict[str, Any]]
```

```
8445:     ) -> dict[str, list[str]]:
8446:         """Extract most relevant text excerpts per dimension"""
8447:         _ = dimension_results  # parameter kept for future compatibility
8448:         excerpts = {}
8449:         for dim, dim_emb in self.dimension_embeddings.items():
8450:             # Rank chunks by similarity
8451:             sims = [
8452:                 (i, ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.PolicyDocumentAnalyzer.analyze", "auto_param_L571_20", 1.0) - cos
ine(chunk["embedding"], dim_emb))
8453:                 for i, chunk in enumerate(chunks)
8454:             ]
8455:             sims.sort(key=lambda x: x[1], reverse=True)
8456:             # Top 3 excerpts
8457:             top_chunks = [chunks[i] for i, _ in sims[:3]]
8458:             excerpts[dim.value] = [
8459:                 _get_chunk_content(c)[:300]
8460:                 + ("..." if len(_get_chunk_content(c)) > 300 else "")
8461:                 for c in top_chunks
8462:             ]
8463:         return excerpts
8464:
8465: # ========================
8466: # PRODUCER CLASS - Registry Exposure
8467: # ========================
8468:
8469: class SemanticChunkingProducer:
8470:     """
8471:     Producer wrapper for semantic chunking and policy analysis with registry exposure
8472:
8473:     Provides public API methods for orchestrator integration without exposing
8474:     internal implementation details or summarization logic.
8475:
8476:     Version: ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.PolicyDocumentAnalyzer.analyze", "auto_param_L595_13", 1.0).0
8477:     Producer Type: Semantic Analysis / Chunking
8478:     """
8479:
8480:     def __init__(self, config: SemanticConfig | None = None) -> None:
8481:         """Initialize producer with optional configuration"""
8482:         self.config = config or SemanticConfig()
8483:         self.semantic = SemanticProcessor(self.config)
8484:         self.bayesian = BayesianEvidenceIntegrator(
8485:             prior_concentration=self.config.bayesian_prior_strength
8486:         )
8487:         self.analyzer = PolicyDocumentAnalyzer(self.config)
8488:         logger.info("SemanticChunkingProducer initialized")
8489:
8490:     # ========================================================================
8491:     # CHUNKING API
8492:     # ========================================================================
8493:
8494:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.chunk_document")
8495:     def chunk_document(self, text: str, preserve_structure: bool = True) -> list[dict[str, Any]]:
8496:         """Chunk document into semantic units with embeddings"""
8497:         return self.semantic.chunk_text(text, preserve_structure)
8498:
8499:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_chunk_count")
```

```
    def get_chunk_count(self, chunks: list[dict[str, Any]]) -> int:
        """Get number of chunks"""
        return len(chunks)

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_chunk_text")
    def get_chunk_text(self, chunk: dict[str, Any]) -> str:
        """Extract text from chunk"""
        return _get_chunk_content(chunk)

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_chunk_embedding")
    def get_chunk_embedding(self, chunk: dict[str, Any]) -> NDArray[np.floating[Any]]:
        """Extract embedding from chunk"""
        return chunk.get("embedding", np.array([]))

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_chunk_metadata")
    def get_chunk_metadata(self, chunk: dict[str, Any]) -> dict[str, Any]:
        """Extract metadata from chunk"""
        return {
            "section_type": chunk.get("section_type"),
            "section_id": chunk.get("section_id"),
            "token_count": chunk.get("token_count"),
            "position": chunk.get("position"),
            "has_table": chunk.get("has_table"),
            "has_numerical": chunk.get("has_numerical")
        }

    # ========================================================================
    # EMBEDDING API
    # ========================================================================

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.embed_text")
    def embed_text(self, text: str) -> NDArray[np.floating[Any]]:
        """Generate single embedding for text"""
        return self.semantic.embed_single(text)

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.embed_batch")
    def embed_batch(self, texts: list[str]) -> list[NDArray[np.floating[Any]]]:
        """Generate embeddings for batch of texts"""
        return self.semantic._embed_batch(texts)

    # ========================================================================
    # ANALYSIS API
    # ========================================================================

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.analyze_document")
    def analyze_document(self, text: str) -> dict[str, Any]:
        """Full pipeline analysis of document"""
        return self.analyzer.analyze(text)

    def get_dimension_analysis(
        self,
        analysis: dict[str, Any],
        dimension: CausalDimension
    ) -> dict[str, Any]:
        """Extract specific dimension results from analysis"""
        return analysis.get("causal_dimensions", {}).get(dimension.value, {})
```

```
 8556:
 8557:     def get_dimension_score(
 8558:         self,
 8559:         analysis: dict[str, Any],
 8560:         dimension: CausalDimension
 8561:     ) -> float:
 8562:         """Extract dimension evidence strength score"""
 8563:         dim_result = self.get_dimension_analysis(analysis, dimension)
 8564:         return dim_result.get("evidence_strength", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.analyze_d
ocument", "auto_param_L683_51", 0.0))
 8565:
 8566:     def get_dimension_confidence(
 8567:         self,
 8568:         analysis: dict[str, Any],
 8569:         dimension: CausalDimension
 8570:     ) -> float:
 8571:         """Extract dimension confidence score"""
 8572:         dim_result = self.get_dimension_analysis(analysis, dimension)
 8573:         return dim_result.get("confidence", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.analyze_document
", "auto_param_L692_44", 0.0))
 8574:
 8575:     def get_dimension_excerpts(
 8576:         self,
 8577:         analysis: dict[str, Any],
 8578:         dimension: CausalDimension
 8579:     ) -> list[str]:
 8580:         """Extract key excerpts for dimension"""
 8581:         return analysis.get("key_excerpts", {}).get(dimension.value, [])
 8582:
 8583:     # ========================================================================
 8584:     # BAYESIAN EVIDENCE API
 8585:     # ========================================================================
 8586:
 8587:     def integrate_evidence(
 8588:         self,
 8589:         similarities: NDArray[np.float64],
 8590:         chunk_metadata: list[dict[str, Any]]
 8591:     ) -> dict[str, float]:
 8592:         """Perform Bayesian evidence integration"""
 8593:         return self.bayesian.integrate_evidence(similarities, chunk_metadata)
 8594:
 8595:     def calculate_causal_strength(
 8596:         self,
 8597:         cause_emb: NDArray[np.floating[Any]],
 8598:         effect_emb: NDArray[np.floating[Any]],
 8599:         context_emb: NDArray[np.floating[Any]]
 8600:     ) -> float:
 8601:         """Calculate causal strength between embeddings"""
 8602:         return self.bayesian.causal_strength(cause_emb, effect_emb, context_emb)
 8603:
 8604:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_mean")
 8605:     def get_posterior_mean(self, evidence: dict[str, float]) -> float:
 8606:         """Extract posterior mean from evidence integration"""
 8607:         return evidence.get("posterior_mean", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_
mean", "auto_param_L726_46", 0.0))
 8608:
```

```
    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_std")
    def get_posterior_std(self, evidence: dict[str, float]) -> float:
        """Extract posterior standard deviation"""
        return evidence.get("posterior_std", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_std", "auto_param_L731_45", 0.0))

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_information_gain")
    def get_information_gain(self, evidence: dict[str, float]) -> float:
        """Extract information gain (KL divergence)"""
        return evidence.get("information_gain", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_information_gain", "auto_param_L736_48", 0.0))

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_confidence")
    def get_confidence(self, evidence: dict[str, float]) -> float:
        """Extract confidence score"""
        return evidence.get("confidence", ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_confidence", "auto_param_L741_42", 0.0))

    # ========================================================================
    # SEMANTIC SEARCH API
    # ========================================================================

    def semantic_search(
        self,
        query: str,
        chunks: list[dict[str, Any]],
        dimension: CausalDimension | None = None,
        top_k: int = 5
    ) -> list[tuple[dict[str, Any], float]]:
        """Search chunks semantically for query"""
        query_emb = self.semantic.embed_single(query)

        results = []
        for chunk in chunks:
            chunk_emb = chunk.get("embedding")
            if chunk_emb is not None and len(chunk_emb) > 0:
                similarity = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_confidence", "auto_param_L761_29", 1.0) - cosine(query_emb, chunk_emb)

                # Filter by dimension if specified
                if dimension is None or chunk.get("section_type") == dimension:
                    results.append((chunk, float(similarity)))

        # Sort by similarity descending
        results.sort(key=lambda x: x[1], reverse=True)

        return results[:top_k]

    # ========================================================================
    # UTILITY API
    # ========================================================================

    @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.list_dimensions")
    def list_dimensions(self) -> list[CausalDimension]:
        """List all causal dimensions"""
        return list(CausalDimension)
```

```
8661:
8662:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_dimension_description")
8663:     def get_dimension_description(self, dimension: CausalDimension) -> str:
8664:         """Get description for dimension"""
8665:         descriptions = {
8666:             CausalDimension.INSUMOS: "Recursos, capacidad institucional",
8667:             CausalDimension.ACTIVIDADES: "Acciones, procesos, cronogramas",
8668:             CausalDimension.PRODUCTOS: "Entregables inmediatos",
8669:             CausalDimension.RESULTADOS: "Efectos mediano plazo",
8670:             CausalDimension.IMPACTOS: "TransformaciÃ³n estructural largo plazo",
8671:             CausalDimension.SUPUESTOS: "Condiciones habilitantes"
8672:         }
8673:         return descriptions.get(dimension, "")
8674:
8675:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.get_config")
8676:     def get_config(self) -> SemanticConfig:
8677:         """Get current configuration"""
8678:         return self.config
8679:
8680:     @calibrated_method("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.set_config")
8681:     def set_config(self, config: SemanticConfig) -> None:
8682:         """Update configuration (requires reinitialization)"""
8683:         self.config = config
8684:         self.semantic = SemanticProcessor(self.config)
8685:         self.bayesian = BayesianEvidenceIntegrator(
8686:             prior_concentration=self.config.bayesian_prior_strength
8687:         )
8688:         self.analyzer = PolicyDocumentAnalyzer(self.config)
8689:
8690: # ========================
8691: # CLI INTERFACE
8692: # ========================
8693:
8694: def main() -> None:
8695:     """Example usage"""
8696:     sample_pdm = """
8697: PLAN DE DESARROLLO MUNICIPAL 2024-2027
8698: MUNICIPIO DE EJEMPLO, COLOMBIA
8699:
8700: 1. DIAGNÃ\223STICO TERRITORIAL
8701: El municipio cuenta con 45,000 habitantes, de los cuales 60% reside en zona rural.
8702: La tasa de pobreza multidimensional es 42.3%, superior al promedio departamental.
8703:
8704: 2. VISIÃ\223N ESTRATÃ\211GICA
8705: Para 2027, el municipio serÃ¡ reconocido por su desarrollo sostenible e inclusivo.
8706:
8707: 3. PLAN PLURIANUAL DE INVERSIONES
8708: Se destinarÃ¡n $12,500 millones al sector educaciÃ³n, con meta de construir
8709: 3 instituciones educativas y capacitar 250 docentes en pedagogÃ-as innovadoras.
8710:
8711: 4. SEGUIMIENTO Y EVALUACIÃ\223N
8712: Se implementarÃ¡ sistema de indicadores alineado con ODS, con mediciones semestrales.
8713: """
8714:     config = SemanticConfig(
8715:         chunk_size=512,
8716:         chunk_overlap=100,
```

```
        similarity_threshold = ParameterLoaderV2.get("farfan_core.processing.semantic_chunking_policy.SemanticChunkingProducer.set_config", "similarity_thre
shold", 0.8) # Refactored
    )
    analyzer = PolicyDocumentAnalyzer(config)
    results = analyzer.analyze(sample_pdm)
    print(json.dumps({
        "summary": results["summary"],
        "dimensions": {
            k: {
                "evidence_strength": v["evidence_strength"],
                "confidence": v["confidence"],
                "information_gain": v["information_gain"]
            }
            for k, v in results["causal_dimensions"].items()
        }
    }, indent=2, ensure_ascii=False))


================================================================================
FILE: src/farfan_pipeline/processing/spc_ingestion.py
================================================================================

"""
SISTEMA INDUSTRIAL SOTA PARA SMART-POLICY-CHUNKS DE PLANES DE DESARROLLO
VERSIÃ\223N 3.0 COMPLETA - SIN PLACEHOLDERS, IMPLEMENTACIÃ\223N TOTAL
FASE 1 DEL PIPELINE: GENERACIÃ\223N DE CHUNKS COMPRENSIVOS, RIGUROSOS Y ESTRATÃ\211GICOS
"""

import os
import re
import logging
import hashlib
import numpy as np
import copy
from dataclasses import dataclass, asdict, field
from typing import Dict, List, Any, Optional, Tuple, Set, Union
from enum import Enum
from pathlib import Path
from scipy.spatial.distance import cosine
from scipy.stats import entropy
from scipy.signal import find_peaks
# Note: torch and transformers imports removed - model lifecycle managed by canonical producers
from datetime import datetime, timezone
from collections import defaultdict, Counter
import json
import networkx as nx
from sklearn.cluster import DBSCAN, AgglomerativeClustering
# Note: cosine_similarity removed - using canonical semantic_search with cross-encoder reranking
from sklearn.decomposition import LatentDirichletAllocation
from sklearn.feature_extraction.text import TfidfVectorizer
import spacy
from nltk.tokenize import sent_tokenize
# Note: SentenceTransformer import removed - embedding handled by canonical producers
import warnings
warnings.filterwarnings('ignore')
```

```
8772:
8773: # ============================================================================
8774: # CANONICAL MODULE INTEGRATION - SOTA Producer APIs
8775: # Import production-grade canonical components from farfan_core.processing
8776: # These replace internal duplicate implementations with frontier SOTA approaches
8777: # ============================================================================
8778:
8779: from farfan_pipeline.processing.embedding_policy import EmbeddingPolicyProducer
8780: from farfan_pipeline.processing.policy_processor import create_policy_processor
8781: from farfan_pipeline.processing.semantic_chunking_policy import SemanticChunkingProducer
8782:
8783: # ============================================================================
8784: # LOGGING CONFIGURADO
8785: # ============================================================================
8786:
8787: logging.basicConfig(
8788:     level=logging.INFO,
8789:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
8790:     handlers=[
8791:         logging.FileHandler('smart_chunks_pipeline.log'),
8792:         logging.StreamHandler()
8793:     ]
8794: )
8795: logger = logging.getLogger("SPC")
8796:
8797: # Optional language detection for multi-language support
8798: try:
8799:     from langdetect import detect
8800:     LANGDETECT_AVAILABLE = True
8801: except ImportError:
8802:     LANGDETECT_AVAILABLE = False
8803:     logger.warning("langdetect not available - defaulting to Spanish models")
8804:
8805: # ============================================================================
8806: # UTILITY FUNCTIONS - Serialization, Hashing, Text Safety
8807: # ============================================================================
8808:
8809: def np_to_list(obj):
8810:     """
8811:     Convert NumPy arrays to lists for JSON serialization.
8812:
8813:     Inputs:
8814:         obj: Any Python object, typically a NumPy array
8815:     Outputs:
8816:         List representation of the array if input is ndarray
8817:     Raises:
8818:         TypeError if object is not JSON-serializable
8819:     """
8820:     if isinstance(obj, np.ndarray):
8821:         return obj.tolist()
8822:     if isinstance(obj, (np.integer, np.floating)):
8823:         return obj.item()
8824:     raise TypeError(f"Type {type(obj)} not serializable")
8825:
8826: def safe_utf8_truncate(text: str, max_bytes: int) -> str:
8827:     """
```

```
    Safely truncate text to max_bytes without cutting multi-byte UTF-8 characters.

    Inputs:
        text (str): Input text to truncate
        max_bytes (int): Maximum number of UTF-8 bytes
    Outputs:
        str: Truncated text that is valid UTF-8
    """
    if not text:
        return text
    encoded = text.encode("utf-8")
    if len(encoded) <= max_bytes:
        return text
    return encoded[:max_bytes].decode("utf-8", "ignore")

def canonical_timestamp() -> str:
    """
    Generate ISO-8601 UTC timestamp with Z suffix for canonical timestamping.

    Inputs:
        None
    Outputs:
        str: ISO-8601 formatted UTC timestamp ending with 'Z'
    """
    return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')

def filter_empty_sentences(sentences: List[str]) -> List[str]:
    """
    Filter out empty or whitespace-only sentences.

    Inputs:
        sentences (List[str]): List of sentence strings
    Outputs:
        List[str]: Filtered list containing only non-empty sentences
    """
    return [s for s in sentences if s.strip()]

# ============================================================================
# CANONICAL ERROR CLASSES
# ============================================================================

class CanonicalError(Exception):
    """Base class for canonical pipeline errors"""
    pass

class ValidationError(CanonicalError):
    """Raised when input validation fails"""
    pass

class ProcessingError(CanonicalError):
    """Raised when processing step fails"""
    pass

class SerializationError(CanonicalError):
    """Raised when serialization fails"""
    pass
```

```
8884:
8885: # ============================================================================
8886: # ENUMS Y TIPOS
8887: # ============================================================================
8888:
8889: class ChunkType(Enum):
8890:     DIAGNOSTICO = "diagnostico"
8891:     ESTRATEGIA = "estrategia"
8892:     METRICA = "metrica"
8893:     FINANCIERO = "financiero"
8894:     NORMATIVO = "normativo"
8895:     OPERATIVO = "operativo"
8896:     EVALUACION = "evaluacion"
8897:     MIXTO = "mixto"
8898:
8899: class CausalRelationType(Enum):
8900:     DIRECT_CAUSE = "direct_cause"
8901:     INDIRECT_CAUSE = "indirect_cause"
8902:     CONDITIONAL = "conditional"
8903:     ENABLING = "enabling"
8904:     PREVENTING = "preventing"
8905:     CORRELATIONAL = "correlational"
8906:     TEMPORAL_PRECEDENCE = "temporal_precedence"
8907:
8908: class PolicyEntityRole(Enum):
8909:     EXECUTOR = "executor"
8910:     BENEFICIARY = "beneficiary"
8911:     REGULATOR = "regulator"
8912:     FUNDER = "funder"
8913:     STAKEHOLDER = "stakeholder"
8914:     EVALUATOR = "evaluator" #
8915:
8916: # ============================================================================
8917: # ESTRUCTURAS DE DATOS
8918: # ============================================================================
8919:
8920: @dataclass
8921: class CausalEvidence:
8922:     dimension: str
8923:     category: str
8924:     matches: List[str]
8925:     confidence: float
8926:     context_span: Tuple[int, int]
8927:     implicit_indicators: List[str]
8928:     causal_type: CausalRelationType
8929:     strength_score: float
8930:     mechanisms: List[str] = field(default_factory=list)
8931:     confounders: List[str] = field(default_factory=list)
8932:     mediators: List[str] = field(default_factory=list)
8933:     moderators: List[str] = field(default_factory=list)
8934:
8935: @dataclass
8936: class PolicyEntity:
8937:     entity_type: str
8938:     text: str
8939:     normalized_form: str
```

```
8940:     context_role: PolicyEntityRole
8941:     confidence: float
8942:     span: Tuple[int, int]
8943:     relationships: List[Tuple[str, str, float]] = field(default_factory=list)
8944:     attributes: Dict[str, Any] = field(default_factory=dict)
8945:     mentioned_count: int = 1
8946:
8947: @dataclass
8948: class Provenance:
8949:     """
8950:     Canonical provenance tracking for policy chunks.
8951:     Ensures 100% traceability to source document page and section.
8952:     """
8953:     page_number: int
8954:     section_header: Optional[str] = None
8955:     bbox: Optional[Tuple[float, float, float, float]] = None  # (x0, y0, x1, y1)
8956:     span_in_page: Optional[Tuple[int, int]] = None
8957:     source_file: Optional[str] = None
8958:
8959: @dataclass
8960: class CrossDocumentReference:
8961:     target_section: str
8962:     reference_type: str
8963:     confidence: float
8964:     semantic_linkage: float
8965:     context_bridge: str
8966:     alignment_score: float = 0.0
8967:     bidirectional: bool = False
8968:     distance_in_doc: int = 0
8969:
8970: @dataclass
8971: class StrategicContext:
8972:     policy_intent: str
8973:     implementation_phase: str
8974:     geographic_scope: str
8975:     temporal_horizon: str
8976:     budget_linkage: str
8977:     risk_factors: List[str]
8978:     success_indicators: List[str]
8979:     alignment_with_sdg: List[str] = field(default_factory=list)
8980:     stakeholder_map: Dict[str, List[str]] = field(default_factory=dict)
8981:     policy_coherence_score: float = 0.0
8982:     intervention_logic_chain: List[str] = field(default_factory=list)
8983:
8984: @dataclass
8985: class ArgumentStructure:
8986:     claims: List[Tuple[str, float]]
8987:     evidence: List[Tuple[str, float]]
8988:     warrants: List[Tuple[str, float]]
8989:     backing: List[Tuple[str, float]]
8990:     rebuttals: List[Tuple[str, float]]
8991:     structure_type: str
8992:     strength_score: float
8993:     logical_coherence: float
8994:
8995: @dataclass
```

```
8996: class TemporalDynamics:
8997:     temporal_markers: List[Tuple[str, str, int]]
8998:     sequence_flow: List[Tuple[str, str, float]]
8999:     dependencies: List[Tuple[str, str, str]]
9000:     milestones: List[Dict[str, Any]]
9001:     temporal_coherence: float
9002:     causality_direction: str
9003:
9004: @dataclass
9005: class SmartPolicyChunk:
9006:     chunk_id: str
9007:     document_id: str
9008:     content_hash: str
9009:     text: str
9010:     normalized_text: str
9011:     semantic_density: float
9012:     section_hierarchy: List[str]
9013:     document_position: Tuple[int, int]
9014:     chunk_type: ChunkType
9015:     causal_chain: List[CausalEvidence]
9016:     policy_entities: List[PolicyEntity]
9017:     implicit_assumptions: List[Tuple[str, float]]
9018:     contextual_presuppositions: List[Tuple[str, float]]
9019:     policy_area_id: str  # PA01-PA10 canonical code
9020:     dimension_id: str    # DIM01-DIM06 canonical code
9021:
9022:     # SOTA Provenance Tracking (optional fields after required)
9023:     provenance: Optional[Provenance] = None
9024:     argument_structure: Optional[ArgumentStructure] = None
9025:     temporal_dynamics: Optional[TemporalDynamics] = None
9026:     discourse_markers: List[Tuple[str, str]] = field(default_factory=list)
9027:     rhetorical_patterns: List[str] = field(default_factory=list)
9028:
9029:     cross_references: List[CrossDocumentReference] = field(default_factory=list)
9030:     strategic_context: Optional[StrategicContext] = None
9031:     related_chunks: List[Tuple[str, float]] = field(default_factory=list)
9032:
9033:     confidence_metrics: Dict[str, float] = field(default_factory=dict)
9034:     coherence_score: float = 0.0
9035:     completeness_index: float = 0.0
9036:     strategic_importance: float = 0.0
9037:     information_density: float = 0.0
9038:     actionability_score: float = 0.0
9039:
9040:     semantic_embedding: Optional[np.ndarray] = None
9041:     policy_embedding: Optional[np.ndarray] = None
9042:     causal_embedding: Optional[np.ndarray] = None
9043:     temporal_embedding: Optional[np.ndarray] = None
9044:
9045:     knowledge_graph_nodes: List[str] = field(default_factory=list)
9046:     knowledge_graph_edges: List[Tuple[str, str, str, float]] = field(default_factory=list)
9047:
9048:     topic_distribution: Dict[str, float] = field(default_factory=dict)
9049:     key_phrases: List[Tuple[str, float]] = field(default_factory=list)
9050:
9051:     processing_timestamp: str = field(default_factory=canonical_timestamp)
```

```
9052:     pipeline_version: str = "SMART-CHUNK-3.0-FINAL"
9053:     extraction_methodology: str = "COMPREHENSIVE_STRATEGIC_ANALYSIS"
9054:     model_versions: Dict[str, str] = field(default_factory=dict) #
9055:
9056:     _CHUNK_ID_PATTERN = re.compile(r"^PA(0[1-9]|10)-DIM(0[1-6])$")
9057:
9058:     def __post_init__(self) -> None:
9059:         """Validate canonical identifiers for irrigation compatibility."""
9060:         if not self.policy_area_id or not self.dimension_id:
9061:             raise ValueError("policy_area_id and dimension_id are required for SmartPolicyChunk")
9062:
9063:         match = self._CHUNK_ID_PATTERN.match(self.chunk_id)
9064:         if not match:
9065:             raise ValueError(
9066:                 f"Invalid chunk_id '{self.chunk_id}'. Expected format PA{{01-10}}-DIM{{01-06}}."
9067:             )
9068:
9069:         pa_code = f"PA{match.group(1)}"
9070:         dim_code = f"DIM{match.group(2)}"
9071:         if pa_code != self.policy_area_id:
9072:             raise ValueError(
9073:                 f"chunk_id {self.chunk_id} mismatches policy_area_id {self.policy_area_id}"
9074:             )
9075:         if dim_code != self.dimension_id:
9076:             raise ValueError(
9077:                 f"chunk_id {self.chunk_id} mismatches dimension_id {self.dimension_id}"
9078:             )
9079:
9080: # ============================================================================
9081: # CONFIGURACIÃ\223N COMPLETA DEL SISTEMA
9082: # ============================================================================
9083:
9084: class SmartChunkConfig:
9085:     # ParÃ¡metros de chunking calibrados
9086:     MIN_CHUNK_SIZE = 300
9087:     MAX_CHUNK_SIZE = 2000
9088:     OPTIMAL_CHUNK_SIZE = 800
9089:     OVERLAP_SIZE = 200
9090:
9091:     # Umbrales semÃ¡nticos
9092:     SEMANTIC_COHERENCE_THRESHOLD = 0.72
9093:     CROSS_REFERENCE_MIN_SIMILARITY = 0.65
9094:     CAUSAL_CHAIN_MIN_CONFIDENCE = 0.60
9095:     ENTITY_EXTRACTION_THRESHOLD = 0.55
9096:
9097:     # ParÃ¡metros de ventana de contexto
9098:     MIN_CONTEXT_WINDOW = 400
9099:     MAX_CONTEXT_WINDOW = 1200
9100:     CONTEXT_EXPANSION_FACTOR = 1.5
9101:
9102:     # Clustering y agrupaciÃ³n
9103:     DBSCAN_EPS = 0.25
9104:     DBSCAN_MIN_SAMPLES = 2
9105:     HIERARCHICAL_CLUSTER_THRESHOLD = 0.70
9106:
9107:     # AnÃ¡lisis causal
```

```
9108:     CAUSAL_CHAIN_MAX_GAP = 3
9109:     TRANSITIVE_CLOSURE_DEPTH = 4
9110:     CAUSAL_MECHANISM_MIN_SUPPORT = 0.50
9111:
9112:     # TÃ³picos y temas
9113:     N_TOPICS_LDA = 15
9114:     MIN_TOPIC_PROBABILITY = 0.15
9115:
9116:     # MÃ©tricas de calidad
9117:     MIN_INFORMATION_DENSITY = 0.40
9118:     MIN_COHERENCE_SCORE = 0.55
9119:     MIN_COMPLETENESS_INDEX = 0.60
9120:     MIN_STRATEGIC_IMPORTANCE = 0.45
9121:
9122:     # DeduplicaciÃ³n
9123:     DEDUPLICATION_THRESHOLD = 0.88
9124:     NEAR_DUPLICATE_THRESHOLD = 0.92 #
9125:
9126: # ============================================================================
9127: # SISTEMAS AUXILIARES COMPLETOS
9128: # ============================================================================
9129:
9130: class ContextPreservationSystem:
9131:     """Sistema de preservaciÃ³n de contexto estratÃ©gico"""
9132:
9133:     def __init__(self, parent_system):
9134:         self.parent = parent_system
9135:         self.logger = logging.getLogger(self.__class__.__name__)
9136:
9137:     def preserve_strategic_context(
9138:         self,
9139:         text: str,
9140:         structural_analysis: Dict,
9141:         global_topics: Dict
9142:     ) -> List[Dict]:
9143:         """
9144:         CANONICAL SOTA: Preserve strategic context using EmbeddingPolicyProducer.
9145:
9146:         Derives breakpoints from canonical chunks with PDM structure awareness.
9147:         Replaces internal breakpoint logic with SOTA semantic chunking.
9148:         """
9149:         # Use canonical chunker with doc_id/title from structural analysis
9150:         chunks = self.parent._spc_embed.process_document(
9151:             text,
9152:             {
9153:                 "doc_id": structural_analysis.get("doc_id", "unknown"),
9154:                 "title": structural_analysis.get("title", "N/A")
9155:             }
9156:         )
9157:
9158:         # Build segments from canonical chunks (position is ordinal; approximate char spans)
9159:         segments = []
9160:         offset = 0
9161:         for ch in chunks:
9162:             ch_text = self.parent._spc_embed.get_chunk_text(ch)
9163:             start = offset
```

```
9164:             end = start + len(ch_text)
9165:             offset = end
9166:
9167:             segment = {
9168:                 "text": ch_text,
9169:                 "context": ch_text,  # Can expand context if needed
9170:                 "position": (start, end),
9171:                 "context_window": (start, end),
9172:                 "semantic_coherence": 0.0,  # Filled by coherence calculation if needed
9173:                 "topic_alignment": self._calculate_topic_alignment(ch_text, global_topics),
9174:                 "pdq_context": self.parent._spc_embed.get_chunk_pdq_context(ch),
9175:                 "metadata": self.parent._spc_embed.get_chunk_metadata(ch),
9176:             }
9177:             segments.append(segment)
9178:
9179:         return segments
9180:
9181:     def _identify_semantic_breakpoints(self, text: str, structural_analysis: Dict) -> List[int]:
9182:         """Identificar puntos de ruptura semÃ¡ntica"""
9183:         breakpoints = [0]
9184:
9185:         # Usar lÃmites de secciÃ³n
9186:         for section in structural_analysis.get('section_hierarchy', []):
9187:             if 'line_number' in section:
9188:                 pos = self._line_to_position(text, section['line_number'])
9189:                 breakpoints.append(pos)
9190:
9191:         # Usar puntos de quiebre estratÃ©gico
9192:         for bp in structural_analysis.get('strategic_breakpoints', []):
9193:             breakpoints.append(bp['position'])
9194:
9195:         # Agregar lÃmites de pÃ¡rrafo significativos
9196:         paragraphs = text.split('\n\n')
9197:         current_pos = 0
9198:         for para in paragraphs:
9199:             if len(para) > self.parent.config.MIN_CHUNK_SIZE:
9200:                 breakpoints.append(current_pos)
9201:             current_pos += len(para) + 2
9202:
9203:         breakpoints.append(len(text))
9204:         return sorted(list(set(breakpoints)))
9205:
9206:     def _line_to_position(self, text: str, line_number: int) -> int:
9207:         """Convertir nÃºmero de lÃnea a posiciÃ³n en texto"""
9208:         lines = text.split('\n')
9209:         position = 0
9210:         for i in range(min(line_number, len(lines))):
9211:             position += len(lines[i]) + 1
9212:         return position
9213:
9214:     def _calculate_segment_coherence(self, segment_text: str) -> float:
9215:         """
9216:         CANONICAL SOTA: Calculate coherence using batch embeddings.
9217:
9218:         Replaces per-sentence embedding calls with efficient batching.
9219:         No per-sentence model churn.
```

```
9220:
9221:         Inputs:
9222:             segment_text (str): Text segment to analyze
9223:         Outputs:
9224:             float: Coherence score between 0.0 and 1.0
9225:         """
9226:         if len(segment_text) < 50:
9227:             return 0.0
9228:
9229:         sentences = filter_empty_sentences(re.split(r'[.!?]+', segment_text))
9230:         if len(sentences) < 2:
9231:             return 0.5
9232:
9233:         # CANONICAL SOTA: Batch embeddings for efficiency
9234:         embs = self.parent._generate_embeddings_for_corpus(sentences, batch_size=64)
9235:
9236:         # Pairwise cosine similarity between consecutive sentences
9237:         sims = np.sum(embs[:-1] * embs[1:], axis=1) / (
9238:             np.linalg.norm(embs[:-1], axis=1) * np.linalg.norm(embs[1:], axis=1) + 1e-8
9239:         )
9240:
9241:         return float(np.mean(sims)) if sims.size else 0.5
9242:
9243:     def _calculate_topic_alignment(self, segment_text: str, global_topics: Dict) -> float:
9244:         """Calcular alineaciÃ³n con tÃ³picos globales"""
9245:         if not global_topics.get('keywords'):
9246:             return 0.5
9247:
9248:         segment_lower = segment_text.lower()
9249:         keyword_matches = 0
9250:
9251:         for keyword, _ in global_topics['keywords'][:20]:
9252:             if keyword.lower() in segment_lower:
9253:                 keyword_matches += 1
9254:
9255:         return min(keyword_matches / 10.0, 1.0) #
9256:
9257:
9258: class CausalChainAnalyzer:
9259:     """Analizador de cadenas causales"""
9260:
9261:     def __init__(self, parent_system):
9262:         self.parent = parent_system
9263:         self.logger = logging.getLogger(self.__class__.__name__)
9264:
9265:     def extract_complete_causal_chains(
9266:         self,
9267:         segments: List[Dict],
9268:         knowledge_graph: Dict
9269:     ) -> List[Dict]:
9270:         """Extraer cadenas causales completas"""
9271:         causal_chains = []
9272:
9273:         for segment in segments:
9274:             chains = self._extract_segment_causal_chains(segment, knowledge_graph)
9275:             causal_chains.extend(chains)
```

```
9276:
9277:         # Conectar cadenas entre segmentos
9278:         connected_chains = self._connect_cross_segment_chains(causal_chains)
9279:
9280:         return connected_chains
9281:
9282:     def _extract_segment_causal_chains(self, segment: Dict, kg: Dict) -> List[Dict]:
9283:         """Extraer cadenas causales de un segmento"""
9284:         text = segment['text']
9285:         chains = []
9286:
9287:         # Patrones causales complejos
9288:         causal_patterns = [
9289:             (r'si\s+([^,]+),\s+entonces\s+([^.]+)', 'conditional'),
9290:             (r'debido\s+a\s+([^,]+),\s+([^.]+)', 'direct_cause'),
9291:             (r'([^,]+)\s+permite\s+([^.]+)', 'enabling'),
9292:             (r'([^,]+)\s+genera\s+([^.]+)', 'generation'),
9293:             (r'para\s+([^,]+),\s+se\s+requiere\s+([^.]+)', 'requirement'),
9294:             (r'([^,]+)\s+resulta\s+en\s+([^.]+)', 'result')
9295:         ]
9296:
9297:         for pattern, chain_type in causal_patterns:
9298:             matches = re.finditer(pattern, text, re.IGNORECASE)
9299:             for match in matches:
9300:                 chain = {
9301:                     'type': chain_type,
9302:                     'antecedent': match.group(1).strip(),
9303:                     'consequent': match.group(2).strip() if match.lastindex >= 2 else '',
9304:                     'position': match.span(),
9305:                     'segment_id': id(segment),
9306:                     'confidence': self._calculate_causal_confidence(match.group(0), text)
9307:                 }
9308:                 chains.append(chain)
9309:
9310:         return chains
9311:
9312:     def _connect_cross_segment_chains(self, chains: List[Dict]) -> List[Dict]:
9313:         """Conectar cadenas causales entre segmentos"""
9314:         if len(chains) < 2:
9315:             return chains
9316:
9317:         # Construir grafo de cadenas
9318:         G = nx.DiGraph()
9319:
9320:         for i, chain in enumerate(chains):
9321:             G.add_node(i, **chain)
9322:
9323:         # Conectar cadenas relacionadas
9324:         for i in range(len(chains)):
9325:             for j in range(i + 1, len(chains)):
9326:                 similarity = self._calculate_chain_similarity(chains[i], chains[j])
9327:                 if similarity > 0.7:
9328:                     G.add_edge(i, j, weight=similarity)
9329:
9330:         # Enriquecer cadenas con conexiones
9331:         for i, chain in enumerate(chains):
```

```
9332:             chain['connections'] = list(G.neighbors(i))
9333:             chain['centrality'] = nx.degree_centrality(G).get(i, 0)
9334:
9335:         return chains
9336:
9337:     def _calculate_causal_confidence(self, match_text: str, context: str) -> float:
9338:         """Calcular confianza de relaciÃ³n causal"""
9339:         confidence = 0.5
9340:
9341:         # Indicadores de confianza alta
9342:         high_confidence_terms = ['garantiza', 'asegura', 'determina', 'causa directamente']
9343:         for term in high_confidence_terms:
9344:             if term in match_text.lower() or term in context.lower():
9345:                 confidence = max(confidence, 0.85)
9346:
9347:         # Indicadores de confianza media
9348:         medium_confidence_terms = ['permite', 'facilita', 'contribuye', 'apoya']
9349:         for term in medium_confidence_terms:
9350:             if term in match_text.lower():
9351:                 confidence = max(confidence, 0.65)
9352:
9353:         # Indicadores de incertidumbre
9354:         uncertainty_terms = ['puede', 'podrÃ-a', 'posiblemente', 'eventualmente']
9355:         for term in uncertainty_terms:
9356:             if term in match_text.lower():
9357:                 confidence = min(confidence, 0.45)
9358:
9359:         return confidence
9360:
9361:     def _calculate_chain_similarity(self, chain1: Dict, chain2: Dict) -> float:
9362:         """
9363:         CANONICAL SOTA: Calculate chain similarity via batch embeddings.
9364:
9365:         Replaces individual embedding calls with efficient batching.
9366:
9367:         Inputs:
9368:             chain1 (Dict): First causal chain
9369:             chain2 (Dict): Second causal chain
9370:         Outputs:
9371:             float: Similarity score between 0.0 and 1.0
9372:         """
9373:         # Build text representations of chains
9374:         texts = [
9375:             f"{chain1.get('antecedent', '')} {chain1.get('consequent', '')}",
9376:             f"{chain2.get('antecedent', '')} {chain2.get('consequent', '')}",
9377:         ]
9378:
9379:         # CANONICAL SOTA: Batch embeddings for efficiency
9380:         embs = self.parent._generate_embeddings_for_corpus(texts, batch_size=2)
9381:
9382:         # Cosine similarity
9383:         sim = float(np.dot(embs[0], embs[1]) / (
9384:             np.linalg.norm(embs[0]) * np.linalg.norm(embs[1]) + 1e-8
9385:         ))
9386:
9387:         # PenalizaciÃ³n por tipo de relaciÃ³n diferente
```

```
        if chain1.get('type') != chain2.get('type'):
            sim *= 0.9

        return sim


class KnowledgeGraphBuilder:
    """Constructor de grafo de conocimiento para polÃ-tica pÃºblica"""

    def __init__(self, parent_system):
        self.parent = parent_system
        self.logger = logging.getLogger(self.__class__.__name__)

    def build_policy_knowledge_graph(self, text: str) -> Dict[str, Any]:
        """Construir grafo de conocimiento de polÃ-tica pÃºblica"""
        G = nx.DiGraph()
        entities = self._extract_all_entities(text)
        relations = self._extract_all_relations(text)
        concepts = self._extract_key_concepts(text)

        # AÃ±adir entidades como nodos
        for entity in entities:
            G.add_node(entity['normalized_form'], type=entity['entity_type'], role=entity['context_role'].value, confidence=entity['confidence'])

        # AÃ±adir relaciones como aristas
        for relation in relations:
            source = self.parent._normalize_entity(relation['source'])
            target = self.parent._normalize_entity(relation['target'])
            if source in G.nodes and target in G.nodes:
                G.add_edge(source, target, type=relation['type'], confidence=relation['confidence'])

        # MÃ©tricas del grafo
        metrics = {
            'num_nodes': G.number_of_nodes(),
            'num_edges': G.number_of_edges(),
            'density': nx.density(G) if G.number_of_nodes() > 0 else 0,
            'components': nx.number_weakly_connected_components(G) if G.number_of_nodes() > 0 else 0
        }

        return {
            'graph': G,
            'entities': entities,
            'relations': relations,
            'concepts': concepts,
            'metrics': metrics
        }

    def _extract_all_entities(self, text: str) -> List[Dict]:
        """Extraer todas las entidades del texto"""
        entities = []

        # Patrones de entidades por tipo
        entity_patterns = {
            'organization': [
                r'(?:Ministerio|SecretarÃ-a|Departamento|DirecciÃ³n|Instituto)\s+(?:de|del?)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©Ã-Ã³Ã°Ã±\s]+',
                r'(?:AlcaldÃ-a|GobernaciÃ³n|Prefectura)\s+(?:de|del?)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©Ã-Ã³Ã°Ã±\s]+'
```

```
            ],
            'program': [
                r'(?:Programa|Plan|Proyecto)\s+(?:de|del?|para)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©Ã-Ã³Ã°Ã±\s]+'
            ],
            'legal_framework': [
                r'(?:Ley|Decreto|ResoluciÃ³n|Acuerdo)\s+No?\s+[\d]+(?: de \d{4})?'
            ]
        }

        for entity_type, patterns in entity_patterns.items():
            for pattern in patterns:
                matches = re.finditer(pattern, text)
                for match in matches:
                    role = self.parent._infer_entity_role(match.group(0), text)
                    entities.append({
                        'text': match.group(0),
                        'normalized_form': self.parent._normalize_entity(match.group(0)),
                        'entity_type': entity_type,
                        'context_role': role,
                        'confidence': 0.8,
                        'span': match.span()
                    })

        return entities

    def _extract_all_relations(self, text: str) -> List[Dict]:
        """Extraer todas las relaciones del texto"""
        relations = []

        # Patrones de relaciones
        relation_patterns = [
            (r'([^,]+)\s+es\s+responsable\s+de\s+([^.]+)', 'responsible_for', 0.9),
            (r'([^,]+)\s+ejecutarÃ¡\s+([^.]+)', 'executes', 0.85),
            (r'([^,]+)\s+beneficiarÃ¡\s+a\s+([^.]+)', 'benefits', 0.8),
            (r'([^,]+)\s+financiarÃ¡\s+([^.]+)', 'funds', 0.85)
        ]

        for pattern, rel_type, confidence in relation_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                if match.lastindex >= 2:
                    relations.append({
                        'source': match.group(1).strip(),
                        'target': match.group(2).strip(),
                        'type': rel_type,
                        'confidence': confidence,
                        'context': match.group(0)
                    })

        return relations

    def _extract_key_concepts(self, text: str) -> List[str]:
        """
        Extract key concepts from noun chunks and key phrases.

        Inputs:
```

```
            text (str): Input text to analyze
        Outputs:
            List[str]: List of key concepts (max 30)
        """
        concepts = []
        if self.parent.nlp:
            # Safe UTF-8 truncation to avoid cutting multi-byte characters
            truncated_text = safe_utf8_truncate(text, 200000)
            doc = self.parent.nlp(truncated_text)
            concepts.extend([chunk.text for chunk in doc.noun_chunks][:50])

        # Normalizar conceptos
        concepts = list(set([c.lower().strip() for c in concepts]))
        return concepts[:30]


class TopicModeler:
    """Modelador de tÃ³picos y temas"""

    def __init__(self, parent_system):
        self.parent = parent_system
        self.logger = logging.getLogger(self.__class__.__name__)
        self.tfidf_vectorizer = TfidfVectorizer(stop_words=self.parent._get_stopwords(), ngram_range=(1, 2), max_df=0.85, min_df=2)
        self.lda_model = LatentDirichletAllocation(n_components=self.parent.config.N_TOPICS_LDA, random_state=42)

    def _get_stopwords(self) -> List[str]:
        """Obtener lista de stopwords en espaÃ±ol (placeholder)"""
        # Una lista de stopwords mÃ¡s completa se usarÃ-a en producciÃ³n
        return ['el', 'la', 'los', 'las', 'un', 'una', 'unos', 'unas', 'y', 'o', 'de', 'a', 'en', 'por', 'con', 'para', 'del', 'al', 'que', 'se', 'es', 'son
', 'han', 'como', 'mÃ¡s', 'pero', 'no', 'su', 'sus']

    def extract_global_topics(self, text_list: List[str]) -> Dict[str, Any]:
        """Extraer tÃ³picos globales mediante LDA"""
        try:
            # 1. Vectorizar
            tfidf_matrix = self.tfidf_vectorizer.fit_transform(text_list)

            # 2. Aplicar LDA
            self.lda_model.fit(tfidf_matrix)
            lda_output = self.lda_model.transform(tfidf_matrix)

            # 3. Extraer tÃ³picos y palabras clave
            feature_names = self.tfidf_vectorizer.get_feature_names_out()
            topics = []

            for topic_idx, topic in enumerate(self.lda_model.components_):
                top_features_ind = topic.argsort()[:-10 - 1:-1]
                top_features = [(feature_names[i], topic[i]) for i in top_features_ind]

                topics.append({
                    'topic_id': topic_idx,
                    'keywords': top_features,
                    'weight': float(topic.sum())
                })

            # Palabras clave globales
```

```
9555:             global_keywords = []
9556:             for topic in topics:
9557:                 global_keywords.extend([kw[0] for kw in topic['keywords']])
9558:             keyword_counts = Counter(global_keywords)
9559:             top_keywords = keyword_counts.most_common(30)
9560:
9561:             return {
9562:                 'topics': topics,
9563:                 'keywords': top_keywords,
9564:                 'topic_distribution': lda_output.mean(axis=0).tolist()
9565:             }
9566:         except Exception as e:
9567:             self.logger.error(f"Error en extracciÃ³n de tÃ³picos: {e}")
9568:             return {'topics': [], 'keywords': []} #
9569:
9570:
9571: class ArgumentAnalyzer:
9572:     """Analizador de estructura argumentativa (Toulmin)"""
9573:
9574:     def __init__(self, parent_system):
9575:         self.parent = parent_system
9576:         self.logger = logging.getLogger(self.__class__.__name__)
9577:
9578:     def analyze_arguments(self, causal_chains: List[Dict]) -> Dict[int, ArgumentStructure]:
9579:         """Analizar argumentos completos"""
9580:         argument_structures = {}
9581:
9582:         # Agrupar cadenas para formar argumentos
9583:         for idx, chain_group in enumerate(self._group_chains_by_proximity(causal_chains)):
9584:             structure = self._extract_argument_structure(chain_group)
9585:             if structure:
9586:                 argument_structures[idx] = structure
9587:
9588:         return argument_structures
9589:
9590:     def _group_chains_by_proximity(self, chains: List[Dict], max_gap: int = 500) -> List[List[Dict]]:
9591:         """Agrupar cadenas causales por proximidad en el texto"""
9592:         if not chains:
9593:             return []
9594:
9595:         chains.sort(key=lambda x: x['position'][0])
9596:         groups = []
9597:         current_group = [chains[0]]
9598:
9599:         for i in range(1, len(chains)):
9600:             prev_end = chains[i-1]['position'][1]
9601:             current_start = chains[i]['position'][0]
9602:
9603:             if current_start - prev_end < max_gap:
9604:                 current_group.append(chains[i])
9605:             else:
9606:                 groups.append(current_group)
9607:                 current_group = [chains[i]]
9608:
9609:         if current_group:
9610:             groups.append(current_group)
```

```
9611:
9612:         return groups
9613:
9614:     def _extract_argument_structure(self, chain_group: List[Dict]) -> Optional[ArgumentStructure]:
9615:         """Extraer los componentes del argumento (Claims, Evidence, Warrants)"""
9616:         if not chain_group:
9617:             return None
9618:
9619:         claims = []
9620:         evidence = []
9621:         warrants = []
9622:
9623:         full_text = ' '.join([f"{c.get('antecedent', '')} {c.get('consequent', '')}" for c in chain_group])
9624:
9625:         # Claims: Resultados (consequents) o afirmaciones directas
9626:         for chain in chain_group:
9627:             if chain.get('type') in ['result', 'generation']:
9628:                 claims.append((chain.get('consequent', ''), chain.get('confidence', 0.5)))
9629:
9630:         # Evidence: Antecedentes o referencias a datos/normas
9631:         for chain in chain_group:
9632:             if chain.get('type') in ['direct_cause', 'conditional', 'requirement']:
9633:                 evidence.append((chain.get('antecedent', ''), chain.get('confidence', 0.5)))
9634:
9635:         # Warrants: Conexiones causales implÃ-citas o explÃ-citas de alta confianza
9636:         warrants.extend(self._identify_warrants(chain_group))
9637:
9638:         # Backing and Rebuttals (Simplificado: Requiere modelo avanzado)
9639:         backing = []
9640:         rebuttals = []
9641:
9642:         # Estructura y Fuerza
9643:         structure_type = self._determine_structure_type(claims, evidence)
9644:         strength_score = self._calculate_argument_strength(claims, evidence, warrants)
9645:         logical_coherence = self._assess_logical_coherence(claims, evidence)
9646:
9647:         return ArgumentStructure(
9648:             claims=claims[:5],
9649:             evidence=evidence[:5],
9650:             warrants=warrants[:3],
9651:             backing=backing,
9652:             rebuttals=rebuttals,
9653:             structure_type=structure_type,
9654:             strength_score=strength_score,
9655:             logical_coherence=logical_coherence
9656:         )
9657:
9658:     def _identify_warrants(self, chain_group: List[Dict]) -> List[Tuple[str, float]]:
9659:         """Identificar warrants (garantÃ-as/conexiones)"""
9660:         warrants = []
9661:         for chain in chain_group:
9662:             if chain.get('confidence', 0.5) > 0.7:
9663:                 warrants.append((f"ConexiÃ³n causal tipo: {chain.get('type')}", chain.get('confidence', 0.5)))
9664:         return warrants
9665:
9666:     def _determine_structure_type(self, claims: List, evidence: List) -> str:
```

```
9667:         """Determinar el tipo de estructura argumentativa"""
9668:         if len(claims) > 1 and len(evidence) >= 1:
9669:             return 'multiple_claims_supported'
9670:         elif len(claims) == 1 and len(evidence) >= 1:
9671:             return 'simple_supported'
9672:         elif not evidence and claims:
9673:             return 'assertion_only'
9674:         elif len(claims) >= 1 and any(c.lower().startswith(('segÃºn', 'de acuerdo con')) for c, _ in evidence):
9675:             return 'evidence_based'
9676:         else:
9677:             return 'balanced'
9678: 
9679:     def _calculate_argument_strength(self, claims: List, evidence: List, warrants: List) -> float:
9680:         """Calcular fuerza del argumento"""
9681:         if not claims:
9682:             return 0.0
9683: 
9684:         claim_strength = np.mean([conf for _, conf in claims]) if claims else 0
9685:         evidence_strength = np.mean([conf for _, conf in evidence]) if evidence else 0
9686:         warrant_strength = np.mean([conf for _, conf in warrants]) if warrants else 0
9687: 
9688:         # PonderaciÃ³n: evidencia mÃ¡s importante que claims
9689:         strength = (claim_strength * 0.3 + evidence_strength * 0.5 + warrant_strength * 0.2)
9690: 
9691:         # Penalizar argumentos sin evidencia
9692:         if not evidence:
9693:             strength *= 0.5
9694: 
9695:         return min(strength, 1.0)
9696: 
9697:     def _assess_logical_coherence(self, claims: List, evidence: List) -> float:
9698:         """Evaluar coherencia lÃ³gica del argumento"""
9699:         if not claims or not evidence:
9700:             return 0.0
9701: 
9702:         # Coherencia basada en similitud semÃ¡ntica entre claims y evidencia
9703:         all_texts = [c for c, _ in claims] + [e for e, _ in evidence]
9704:         if len(all_texts) < 2:
9705:             return 0.5
9706: 
9707:         # Use batch embedding for efficiency
9708:         embeddings = self.parent._generate_embeddings_for_corpus(all_texts, batch_size=64)
9709: 
9710:         # Vectorized cosine similarity computation (no sklearn dependency)
9711:         # Normalize embeddings for efficient dot product = cosine similarity
9712:         norms = np.linalg.norm(embeddings, axis=1, keepdims=True)
9713:         normalized_embs = embeddings / (norms + 1e-8)
9714:         sim_matrix = np.dot(normalized_embs, normalized_embs.T)
9715: 
9716:         # Tomar la similitud media (excluyendo la diagonal)
9717:         coherence = (np.sum(sim_matrix) - np.trace(sim_matrix)) / (len(sim_matrix)**2 - len(sim_matrix))
9718: 
9719:         return min(max(coherence, 0.0), 1.0)
9720: 
9721: 
9722: class TemporalAnalyzer:
```

```
    """Analizador de dinÃ¡mica temporal y secuencial"""

    def __init__(self, parent_system):
        self.parent = parent_system
        self.logger = logging.getLogger(self.__class__.__name__)

    def analyze_temporal_dynamics(self, causal_chains: List[Dict]) -> Dict[int, Optional[TemporalDynamics]]:
        """Analizar dinÃ¡mica temporal completa"""
        temporal_structures = {}

        # Agrupar cadenas para anÃ¡lisis temporal
        for idx, chain_group in enumerate(self.parent.argument_analyzer._group_chains_by_proximity(causal_chains)):
            structure = self._extract_temporal_structure(chain_group)
            if structure:
                temporal_structures[idx] = structure

        return temporal_structures

    def _extract_temporal_structure(self, chain_group: List[Dict]) -> Optional[TemporalDynamics]:
        """Extraer marcadores, secuencias y dependencias temporales"""
        if not chain_group:
            return None

        temporal_markers = []
        sequence_flow = []
        dependencies = []
        milestones = []

        # Extraer marcadores de tiempo
        for chain in chain_group:
            # Reutilizar el analizador temporal de la clase principal
            text_context = f"{chain.get('antecedent', '')} {chain.get('consequent', '')}"
            temp_info = self.parent._analyze_temporal_structure(text_context)

            for marker in temp_info.get('time_markers', []):
                temporal_markers.append((marker['text'], marker['type'], marker['position'][0]))

            for seq in temp_info.get('sequences', []):
                sequence_flow.append((seq['marker'], str(seq['order']), 0.8))

        # Extraer dependencias causales con implicaciÃ³n temporal
        for chain in chain_group:
            if chain.get('type') in ['conditional', 'direct_cause', 'requirement']:
                dependencies.append((
                    chain.get('antecedent', ''),
                    chain.get('consequent', ''),
                    'prerequisite' if chain.get('type') == 'requirement' else 'temporal_precedence'
                ))
            elif chain.get('type') == 'conditional':
                dependencies.append((
                    chain.get('antecedent', ''),
                    chain.get('consequent', ''),
                    'conditional'
                ))

        # Extraer hitos
```

```
9779:         milestone_patterns = [
9780:             r'meta\s+(?:de|para)\s+([^.]+)',
9781:             r'lograr\s+([^.]+)\s+(?:en|para)\s+(20\d{2})',
9782:             r'alcanzar\s+([^.]+)'
9783:         ]
9784:
9785:         for chain in chain_group:
9786:             text = f"{chain.get('antecedent', '')} {chain.get('consequent', '')}"
9787:             for pattern in milestone_patterns:
9788:                 matches = re.findall(pattern, text, re.IGNORECASE)
9789:                 for match in matches[:2]:
9790:                     milestones.append({
9791:                         'description': match if isinstance(match, str) else match[0],
9792:                         'target_date': match[1] if isinstance(match, tuple) and len(match) > 1 else None,
9793:                         'confidence': chain.get('confidence', 0.5)
9794:                     })
9795:
9796:         if not (temporal_markers or sequence_flow or dependencies or milestones):
9797:             return None
9798:
9799:         temporal_coherence = self._calculate_temporal_coherence(temporal_markers, sequence_flow)
9800:         causality_direction = self._determine_causality_direction(dependencies)
9801:
9802:         return TemporalDynamics(
9803:             temporal_markers=temporal_markers[:10],
9804:             sequence_flow=sequence_flow[:5],
9805:             dependencies=dependencies[:10],
9806:             milestones=milestones[:5],
9807:             temporal_coherence=temporal_coherence,
9808:             causality_direction=causality_direction
9809:         )
9810:
9811:     def _calculate_temporal_coherence(self, markers: List, flow: List) -> float:
9812:         """Calcular la coherencia de los marcadores temporales"""
9813:         # Simple mÃ©trica basada en la presencia de orden y hitos
9814:         score = 0.0
9815:         if flow:
9816:             score += 0.5
9817:         if any(m[1] in ['year', 'month_year', 'period'] for m in markers):
9818:             score += 0.5
9819:         return min(score, 1.0)
9820:
9821:     def _determine_causality_direction(self, dependencies: List[Tuple]) -> str:
9822:         """Determinar la direcciÃ³n dominante de la causalidad (forward/backward)"""
9823:         forward = sum(1 for _, _, t in dependencies if t == 'temporal_precedence')
9824:         backward = sum(1 for _, _, t in dependencies if t == 'backward')
9825:
9826:         if forward > backward:
9827:             return 'forward'
9828:         elif backward > forward:
9829:             return 'backward'
9830:         return 'mixed' #
9831:
9832:
9833: class DiscourseAnalyzer:
9834:     """Analizador de discurso"""
```

```
9835:
9836:     def __init__(self, parent_system):
9837:         self.parent = parent_system
9838:         self.logger = logging.getLogger(self.__class__.__name__)
9839:
9840:     def analyze_discourse(self, causal_chains: List[Dict]) -> Dict[int, Dict]:
9841:         """Analizar estructuras discursivas"""
9842:         discourse_structures = {}
9843:
9844:         for idx, chain_group in enumerate(self.parent.argument_analyzer._group_chains_by_proximity(causal_chains, max_gap=300)):
9845:             structure = self._extract_discourse_structure(chain_group)
9846:             if structure:
9847:                 discourse_structures[idx] = structure
9848:
9849:         return discourse_structures
9850:
9851:     def _group_chains_discursively(self, chains: List[Dict]) -> List[List[Dict]]:
9852:         """Agrupar cadenas por coherencia discursiva (reutiliza lÃ³gica de ArgumentAnalyzer)"""
9853:         return self.parent.argument_analyzer._group_chains_by_proximity(chains, max_gap=300)
9854:
9855:     def _extract_discourse_structure(self, chain_group: List[Dict]) -> Optional[Dict]:
9856:         """Extraer estructura discursiva del grupo"""
9857:         if not chain_group:
9858:             return None
9859:
9860:         full_text = ' '.join([f"{c.get('antecedent', '')} {c.get('consequent', '')}" for c in chain_group])
9861:
9862:         # AnÃ¡lisis de relaciones
9863:         relations = self.parent._extract_coherence_relations(full_text)
9864:
9865:         # AnÃ¡lisis retÃ³rico
9866:         rhetorical = self.parent._analyze_rhetorical_structure(full_text)
9867:
9868:         # AnÃ¡lisis de flujo de informaciÃ³n
9869:         info_flow = self.parent._analyze_information_flow(full_text)
9870:
9871:         return {
9872:             'coherence_relations': relations[:10],
9873:             'rhetorical_moves': list(rhetorical.keys()),
9874:             'flow_metrics': info_flow,
9875:             'complexity_score': self._calculate_discourse_complexity(relations, rhetorical)
9876:         }
9877:
9878:     def _calculate_discourse_complexity(self, relations: List[Dict], rhetorical: Dict) -> float:
9879:         """Calcular complejidad discursiva"""
9880:         moves = []
9881:         for v in rhetorical.values():
9882:             moves.extend(v)
9883:
9884:         move_diversity = len(set(moves)) if moves else 0
9885:         relation_diversity = len(set(r['type'] for r in relations)) if relations else 0
9886:
9887:         complexity = (move_diversity / 5.0) * 0.5 + (relation_diversity / 8.0) * 0.5
9888:         return min(complexity, 1.0) #
9889:
9890:
```

```
class StrategicIntegrator:
    """Integrador de anÃ¡lisis multi-escala"""

    def __init__(self, parent_system):
        self.parent = parent_system
        self.logger = logging.getLogger(self.__class__.__name__)

    def integrate_strategic_units(
        self,
        causal_chains: List[Dict],
        structural_analysis: Dict,
        argument_structures: Dict,
        temporal_structures: Dict,
        discourse_structures: Dict,
        global_topics: Dict,
        global_kg: Dict
    ) -> List[Dict]:
        """Integrar todos los anÃ¡lisis en unidades estratÃ©gicas"""

        # 1. Agrupar cadenas en unidades base (reutilizando la agrupaciÃ³n)
        grouped_chains = self.parent.argument_analyzer._group_chains_by_proximity(causal_chains, max_gap=100)

        strategic_units = []

        for idx, chain_group in enumerate(grouped_chains):
            full_text = ' '.join([f"{c.get('antecedent', '')} {c.get('consequent', '')}" for c in chain_group])
            # Derive hierarchy for this segment
            start_pos = chain_group[0]['position'][0] if chain_group and 'position' in chain_group[0] else 0
            hierarchy = self._derive_hierarchy_for_segment(start_pos, structural_analysis)
            # Unidades de integraciÃ³n
            unit = {
                'index': idx,
                'text': full_text,
                'position': (chain_group[0]['position'][0], chain_group[-1]['position'][1]),
                'chains': chain_group,
                'argument_structure': argument_structures.get(idx),
                'temporal_dynamics': temporal_structures.get(idx),
                'discourse_structure': discourse_structures.get(idx),
                'hierarchy': hierarchy,
                'confidence': np.mean([c.get('confidence', 0.5) for c in chain_group]),
                'semantic_coherence': self.parent.context_preserver._calculate_segment_coherence(full_text)
            }

            strategic_units.append(unit)

        # 2. Enriquecer con metadatos estratÃ©gicos
        enriched_units = self._enrich_strategic_metadata(strategic_units)

        # 3. Refinar lÃmites de las unidades (placeholder para refinamiento avanzado)

        return enriched_units

    def _derive_hierarchy_for_segment(self, start_pos: int, structural_analysis: Dict) -> List[str]:
        """Derivar la jerarquÃa de secciÃ³n para una posiciÃ³n en el texto"""
        hierarchy = []
        best_match = None
```

```
        min_distance = float('inf')

        for section in structural_analysis.get('section_hierarchy', []):
            sec_pos = self.parent.context_preserver._line_to_position(structural_analysis['raw_text'], section['line_number'])
            distance = start_pos - sec_pos

            # La secciÃ³n debe preceder o estar en la unidad
            if distance >= -50 and distance < min_distance:
                best_match = section
                min_distance = distance

        if best_match:
            hierarchy.append(best_match['title'])

        return hierarchy

    def _enrich_strategic_metadata(self, units: List[Dict]) -> List[Dict]:
        """Enriquecer unidades con metadatos estratÃ©gicos"""
        for unit in units:
            unit['strategic_weight'] = self._calculate_strategic_weight(unit)
            unit['implementation_readiness'] = self._assess_implementation_readiness(unit)
            unit['risk_level'] = self._assess_risk_level(unit)
        return units

    def _calculate_strategic_weight(self, unit: Dict) -> float:
        """Calcular peso estratÃ©gico de la unidad"""
        factors = {
            'chain_count': min(len(unit['chains']) / 5, 1.0),
            'confidence': unit['confidence'],
            'coherence': unit['semantic_coherence'],
            'hierarchy_level': 1.0 if unit['hierarchy'] else 0.5
        }
        return np.mean(list(factors.values()))

    def _assess_implementation_readiness(self, unit: Dict) -> float:
        """Evaluar preparaciÃ³n para implementaciÃ³n"""
        text = unit.get('text', '')
        readiness_indicators = [
            'plan de acciÃ³n', 'presupuesto asignado', 'cronograma definido',
            'responsable designado', 'indicadores de seguimiento'
        ]
        readiness_score = sum(1 for ind in readiness_indicators if ind in text.lower())
        return min(readiness_score / 3.0, 1.0)

    def _assess_risk_level(self, unit: Dict) -> float:
        """Evaluar nivel de riesgo (simplificado)"""
        text = unit.get('text', '')
        risk_terms = ['riesgo', 'limitaciÃ³n', 'desafÃ-o', 'obstÃ¡culo', 'incertidumbre']
        risk_score = sum(1 for term in risk_terms if term in text.lower())
        return min(risk_score / 3.0, 1.0) #


# ==============================================================================
# POLICY AREA CHUNK CALIBRATION - Garantiza 10 chunks por policy area
# ==============================================================================

```

```
class PolicyAreaChunkCalibrator:
    """
    Calibrates chunking to guarantee exactly 10 strategic chunks per policy area.

    Uses the existing SemanticChunkingProducer with dynamically adjusted parameters
    to ensure consistent chunk count across different policy documents.

    Strategy:
        1. Estimate optimal chunk_size based on document length
        2. Generate initial chunks with SemanticChunkingProducer
        3. Adjust parameters iteratively if chunk count != 10
        4. Merge or split chunks as needed to reach target

    Attributes:
        TARGET_CHUNKS_PER_PA: Target number of chunks (10)
        TOLERANCE: Acceptable deviation (Â±1 chunk)
        MAX_ITERATIONS: Maximum calibration iterations (3)
    """

    TARGET_CHUNKS_PER_PA = 10
    TOLERANCE = 1
    MAX_ITERATIONS = 3

    # Canonical policy areas from questionnaire_monolith.json
    POLICY_AREAS = [
        "PA01", "PA02", "PA03", "PA04", "PA05",
        "PA06", "PA07", "PA08", "PA09", "PA10"
    ]

    def __init__(self, semantic_chunking_producer: SemanticChunkingProducer):
        """
        Initialize calibrator.

        Args:
            semantic_chunking_producer: Canonical SemanticChunkingProducer instance
        """
        self.chunking_producer = semantic_chunking_producer
        self.logger = logging.getLogger("SPC.Calibrator")

    def calibrate_for_policy_area(
        self,
        text: str,
        policy_area: str,
        metadata: Optional[Dict[str, Any]] = None
    ) -> List[Dict[str, Any]]:
        """
        Generate exactly 10 chunks for a policy area.

        Args:
            text: Policy document text
            policy_area: Policy area ID (PA01-PA10)
            metadata: Optional metadata to attach to chunks

        Returns:
            List of exactly 10 chunks

```

```
        Raises:
            ValueError: If policy_area is invalid or text is empty
        """
        if policy_area not in self.POLICY_AREAS:
            raise ValueError(
                f"Invalid policy_area: {policy_area}. "
                f"Must be one of {self.POLICY_AREAS}"
            )

        if not text or len(text.strip()) == 0:
            raise ValueError("Text cannot be empty")

        self.logger.info(
            f"Calibrating chunks for {policy_area} "
            f"(target: {self.TARGET_CHUNKS_PER_PA} chunks)"
        )

        # Estimate initial parameters based on document length
        initial_params = self._estimate_initial_params(text)

        # Attempt to generate chunks with calibration
        chunks = self._generate_with_calibration(
            text,
            policy_area,
            initial_params,
            metadata
        )

        # Final validation
        if len(chunks) != self.TARGET_CHUNKS_PER_PA:
            self.logger.warning(
                f"{policy_area}: Could not reach exact target. "
                f"Got {len(chunks)} chunks, forcing adjustment to {self.TARGET_CHUNKS_PER_PA}"
            )
            chunks = self._force_chunk_count(chunks, self.TARGET_CHUNKS_PER_PA)

        self.logger.info(
            f"{policy_area}: Calibration complete - {len(chunks)} chunks generated"
        )

        return chunks

    def _estimate_initial_params(self, text: str) -> Dict[str, Any]:
        """
        Estimate optimal chunking parameters based on text length.

        Args:
            text: Document text

        Returns:
            Dictionary with chunk_size, overlap, and other parameters
        """
        text_length = len(text)
        sentence_count = text.count('.') + text.count('!') + text.count('?')

        # Estimate chunk size to yield ~10 chunks
```

```
10115:         estimated_chunk_size = max(
10116:             500,  # Minimum chunk size
10117:             min(
10118:                 2000,  # Maximum chunk size
10119:                 text_length // (self.TARGET_CHUNKS_PER_PA + 2)  # Add buffer
10120:             )
10121:         )
10122:
10123:         return {
10124:             'chunk_size': estimated_chunk_size,
10125:             'overlap': int(estimated_chunk_size * 0.15),  # 15% overlap
10126:             'min_chunk_size': 300,
10127:             'adaptive': True,
10128:         }
10129:
10130:     def _generate_with_calibration(
10131:         self,
10132:         text: str,
10133:         policy_area: str,
10134:         initial_params: Dict[str, Any],
10135:         metadata: Optional[Dict[str, Any]]
10136:     ) -> List[Dict[str, Any]]:
10137:         """
10138:         Generate chunks with iterative calibration to reach target count.
10139:
10140:         Args:
10141:             text: Document text
10142:             policy_area: Policy area ID
10143:             initial_params: Initial chunking parameters
10144:             metadata: Optional metadata
10145:
10146:         Returns:
10147:             List of chunks (may not be exactly 10, needs final adjustment)
10148:         """
10149:         params = initial_params.copy()
10150:
10151:         for iteration in range(self.MAX_ITERATIONS):
10152:             # Use SemanticChunkingProducer to generate chunks
10153:             try:
10154:                 # FIXED: Actually use the SemanticChunkingProducer
10155:                 chunks = self._generate_chunks_with_producer(text, params, policy_area, metadata)
10156:             except Exception as e:
10157:                 # Fallback to simple chunking if producer fails
10158:                 self.logger.warning(f"SemanticChunkingProducer failed: {e}, using fallback")
10159:                 chunks = self._generate_chunks_simple(text, params, policy_area, metadata)
10160:
10161:             chunk_count = len(chunks)
10162:             delta = chunk_count - self.TARGET_CHUNKS_PER_PA
10163:
10164:             self.logger.debug(
10165:                 f"{policy_area}: Iteration {iteration+1} - "
10166:                 f"{chunk_count} chunks (delta: {delta:+d})"
10167:             )
10168:
10169:             # Check if within tolerance
10170:             if abs(delta) <= self.TOLERANCE:
```

```
10171:                 return chunks
10172:
10173:             # Adjust parameters for next iteration
10174:             if delta > 0:
10175:                 # Too many chunks - increase chunk size
10176:                 params['chunk_size'] = int(params['chunk_size'] * 1.2)
10177:             else:
10178:                 # Too few chunks - decrease chunk size
10179:                 params['chunk_size'] = int(params['chunk_size'] * 0.8)
10180:
10181:             # Ensure bounds
10182:             params['chunk_size'] = max(400, min(2500, params['chunk_size']))
10183:
10184:         # Return best attempt after max iterations
10185:         return chunks
10186:
10187:     def _generate_chunks_with_producer(
10188:         self,
10189:         text: str,
10190:         params: Dict[str, Any],
10191:         policy_area: str,
10192:         metadata: Optional[Dict[str, Any]]
10193:     ) -> List[Dict[str, Any]]:
10194:         """
10195:         Generate chunks using the SemanticChunkingProducer.
10196:
10197:         Args:
10198:             text: Document text
10199:             params: Chunking parameters (chunk_size, overlap, etc.)
10200:             policy_area: Policy area ID
10201:             metadata: Optional metadata
10202:
10203:         Returns:
10204:             List of chunk dictionaries
10205:         """
10206:         # Use the actual SemanticChunkingProducer (instance method, not standalone function)
10207:         # Use the producer instance injected via __init__
10208:         producer = self.chunking_producer
10209:
10210:         # chunk_document signature: (text: str, preserve_structure: bool = True) -> list[dict[str, Any]]
10211:         result_chunks = producer.chunk_document(text=text, preserve_structure=True)
10212:
10213:         # Convert to our format
10214:         chunks = []
10215:         for i, chunk_result in enumerate(result_chunks):
10216:             # chunk_result is a dict with keys like 'text', 'embedding', 'section_type', etc.
10217:             chunks.append({
10218:                 'id': f"{policy_area}_chunk_{i+1}",
10219:                 'text': chunk_result.get('text', ''),
10220:                 'policy_area': policy_area,
10221:                 'chunk_index': i,
10222:                 'length': len(chunk_result.get('text', '')),
10223:                 'metadata': metadata or {},
10224:                 'semantic_metadata': {
10225:                     'section_type': chunk_result.get('section_type'),
10226:                     'section_id': chunk_result.get('section_id'),
```

```
                    'has_embedding': 'embedding' in chunk_result
                }
            })

        return chunks

    def _generate_chunks_simple(
        self,
        text: str,
        params: Dict[str, Any],
        policy_area: str,
        metadata: Optional[Dict[str, Any]]
    ) -> List[Dict[str, Any]]:
        """
        Simple chunk generation using sentence splitting.

        This is a fallback implementation. In production, this would use
        the full SemanticChunkingProducer with BGE-M3 embeddings.

        Args:
            text: Document text
            params: Chunking parameters
            policy_area: Policy area ID
            metadata: Optional metadata

        Returns:
            List of chunk dictionaries
        """
        # Simple sentence-based chunking
        sentences = re.split(r'[.!?]+', text)
        sentences = [s.strip() for s in sentences if s.strip()]

        chunk_size = params.get('chunk_size', 1000)
        chunks = []
        current_chunk = []
        current_length = 0

        for sentence in sentences:
            sentence_length = len(sentence)

            if current_length + sentence_length > chunk_size and current_chunk:
                # Create chunk
                chunk_text = '. '.join(current_chunk) + '.'
                chunks.append({
                    'id': f"{policy_area}_chunk_{len(chunks)+1}",
                    'text': chunk_text,
                    'policy_area': policy_area,
                    'chunk_index': len(chunks),
                    'length': len(chunk_text),
                    'metadata': metadata or {}
                })
                current_chunk = [sentence]
                current_length = sentence_length
            else:
                current_chunk.append(sentence)
                current_length += sentence_length
```

```
10283:
10284:         # Add final chunk
10285:         if current_chunk:
10286:             chunk_text = '. '.join(current_chunk) + '.'
10287:             chunks.append({
10288:                 'id': f"{policy_area}_chunk_{len(chunks)+1}",
10289:                 'text': chunk_text,
10290:                 'policy_area': policy_area,
10291:                 'chunk_index': len(chunks),
10292:                 'length': len(chunk_text),
10293:                 'metadata': metadata or {}
10294:             })
10295:
10296:         return chunks
10297:
10298:     def _force_chunk_count(
10299:         self,
10300:         chunks: List[Dict[str, Any]],
10301:         target: int
10302:     ) -> List[Dict[str, Any]]:
10303:         """
10304:         Force chunk count to exactly match target by merging or splitting.
10305:
10306:         Args:
10307:             chunks: List of chunks
10308:             target: Target chunk count
10309:
10310:         Returns:
10311:             List with exactly target chunks
10312:         """
10313:         current_count = len(chunks)
10314:
10315:         if current_count == target:
10316:             return chunks
10317:
10318:         if current_count > target:
10319:             # Too many chunks - merge smallest adjacent pairs
10320:             while len(chunks) > target:
10321:                 # Find smallest chunk
10322:                 min_idx = min(range(len(chunks)), key=lambda i: chunks[i]['length'])
10323:
10324:                 # Merge with adjacent chunk
10325:                 if min_idx > 0:
10326:                     # Merge with previous
10327:                     chunks[min_idx-1]['text'] += ' ' + chunks[min_idx]['text']
10328:                     chunks[min_idx-1]['length'] = len(chunks[min_idx-1]['text'])
10329:                     chunks.pop(min_idx)
10330:                 else:
10331:                     # Merge with next
10332:                     chunks[min_idx]['text'] += ' ' + chunks[min_idx+1]['text']
10333:                     chunks[min_idx]['length'] = len(chunks[min_idx]['text'])
10334:                     chunks.pop(min_idx+1)
10335:         else:
10336:             # Too few chunks - split largest chunks
10337:             while len(chunks) < target:
10338:                 # Find largest chunk
```

```
10339:                 max_idx = max(range(len(chunks)), key=lambda i: chunks[i]['length'])
10340:
10341:                 # Split it in half
10342:                 chunk_to_split = chunks[max_idx]
10343:                 text = chunk_to_split['text']
10344:                 mid_point = len(text) // 2
10345:
10346:                 # Find sentence boundary near midpoint
10347:                 split_point = text.rfind('.', 0, mid_point) + 1
10348:                 if split_point <= 0:
10349:                     split_point = mid_point
10350:
10351:                 # Create two chunks
10352:                 chunk1_text = text[:split_point].strip()
10353:                 chunk2_text = text[split_point:].strip()
10354:
10355:                 chunks[max_idx] = {
10356:                     **chunk_to_split,
10357:                     'text': chunk1_text,
10358:                     'length': len(chunk1_text),
10359:                 }
10360:
10361:                 chunks.insert(max_idx + 1, {
10362:                     **chunk_to_split,
10363:                     'id': f"{chunk_to_split['id']}_split",
10364:                     'text': chunk2_text,
10365:                     'length': len(chunk2_text),
10366:                 })
10367:
10368:         # Re-index chunks
10369:         for i, chunk in enumerate(chunks):
10370:             chunk['chunk_index'] = i
10371:
10372:         return chunks
10373:
10374:
10375: # ============================================================================
10376: # SISTEMA PRINCIPAL DE CHUNKING ESTRATÃ\211GICO (COMPLETO)
10377: # ============================================================================
10378:
10379: class StrategicChunkingSystem:
10380:     def __init__(self, random_seed: int = 42):
10381:         """
10382:         Initialize the Strategic Chunking System with canonical components.
10383:
10384:         Integrates production-grade canonical modules:
10385:         - PolicyAnalysisEmbedder for semantic embeddings
10386:         - SemanticProcessor for chunking with PDM structure awareness
10387:         - IndustrialPolicyProcessor for causal evidence extraction
10388:         - BayesianEvidenceScorer for probabilistic confidence scoring
10389:
10390:         Args:
10391:             random_seed: Seed for deterministic RNG (default: 42)
10392:
10393:         Inputs:
10394:             None
```

```
10395:         Outputs:
10396:             None - initializes system state
10397:         """
10398:         # Fix seeds for deterministic execution (HOSTILE AUDIT REQUIREMENT)
10399:         import random
10400:         np.random.seed(random_seed)
10401:         random.seed(random_seed)
10402:         self.logger = logging.getLogger("SPC")  # Unified logger name
10403:         self.logger.info(f"Initialized with deterministic seed: {random_seed}")
10404:
10405:         self.config = SmartChunkConfig()
10406:
10407:         # ========================================================================
10408:         # CANONICAL SOTA PRODUCERS - Frontier approach components
10409:         # ========================================================================
10410:
10411:         # Initialize SOTA canonical producers that replace internal implementations
10412:         # These provide BGE-M3 embeddings, cross-encoder reranking, Bayesian numerical eval
10413:         self._spc_embed = EmbeddingPolicyProducer()          # chunking + embeddings + search + Bayesian numeric eval
10414:         self._spc_sem = SemanticChunkingProducer()           # direct embed_text/embed_batch and chunk_document
10415:         self._spc_policy = create_policy_processor()         # canonical PDQ/dimension evidence
10416:
10417:         self.logger.info("SOTA canonical producers initialized: EmbeddingPolicyProducer, SemanticChunkingProducer, PolicyProcessor")
10418:
10419:         # ========================================================================
10420:         # POLICY AREA Ã\227 DIMENSION KEYWORD MAPS - Structured extraction
10421:         # ========================================================================
10422:
10423:         # PA01-PA10 keyword maps for content extraction
10424:         self._pa_keywords = {
10425:             "PA01": ["mujeres", "gÃ©nero", "igualdad", "feminismo", "violencia gÃ©nero", "empoderamiento", "equidad"],
10426:             "PA02": ["violencia", "conflicto armado", "protecciÃ³n", "prevenciÃ³n", "grupos delincuenciales", "economÃ-as ilegales", "seguridad"],
10427:             "PA03": ["ambiente", "cambio climÃ¡tico", "desastres", "medio ambiente", "ecologÃ-a", "sostenibilidad", "recursos naturales"],
10428:             "PA04": ["derechos econÃ³micos", "derechos sociales", "derechos culturales", "educaciÃ³n", "salud", "vivienda", "trabajo"],
10429:             "PA05": ["vÃ-ctimas", "construcciÃ³n de paz", "reconciliaciÃ³n", "reparaciÃ³n", "memoria", "justicia transicional"],
10430:             "PA06": ["niÃ±ez", "adolescencia", "juventud", "entornos protectores", "desarrollo infantil", "educaciÃ³n inicial"],
10431:             "PA07": ["tierras", "territorios", "tenencia", "reforma agraria", "ordenamiento territorial", "catastro"],
10432:             "PA08": ["lÃ-deres", "lideresas", "defensores", "defensoras", "derechos humanos", "protecciÃ³n lÃ-deres", "amenazas"],
10433:             "PA09": ["privadas libertad", "cÃ¡rceles", "sistema penitenciario", "hacinamiento", "reinserciÃ³n", "reclusos"],
10434:             "PA10": ["migraciÃ³n", "transfronteriza", "migrantes", "refugiados", "movilidad humana", "frontera"]
10435:         }
10436:
10437:         # DIM01-DIM06 keyword maps for dimension alignment
10438:         self._dim_keywords = {
10439:             "DIM01": ["diagnÃ³stico", "recursos", "presupuesto", "financiaciÃ³n", "insumos", "inversiÃ³n", "dotaciÃ³n"],
10440:             "DIM02": ["actividades", "intervenciÃ³n", "diseÃ±o", "estrategias", "acciones", "programas", "proyectos"],
10441:             "DIM03": ["productos", "outputs", "entregables", "resultados intermedios", "metas", "indicadores producto"],
10442:             "DIM04": ["resultados", "outcomes", "efectos", "logros", "cambios", "impacto directo", "beneficiarios"],
10443:             "DIM05": ["impactos", "largo plazo", "transformaciÃ³n", "cambio estructural", "sostenibilidad", "legado"],
10444:             "DIM06": ["causalidad", "teorÃ-a de cambio", "cadena causal", "lÃ³gica intervenciÃ³n", "marco lÃ³gico", "supuestos"]
10445:         }
10446:
10447:         self.logger.info(f"PA keyword maps: {len(self._pa_keywords)} policy areas")
10448:         self.logger.info(f"DIM keyword maps: {len(self._dim_keywords)} dimensions")
10449:
10450:         # ========================================================================
```

```
10451:         # SPECIALIZED COMPONENTS - Keep (no canonical equivalent)
10452:         # ========================================================================
10453:
10454:         # These provide unique Smart Policy Chunks innovations
10455:         self._nlp = None  # SpaCy for NER (lazy-loaded)
10456:         self._kg_builder = None  # NetworkX knowledge graph
10457:         self._topic_modeler = None  # LDA topic modeling
10458:         self._argument_analyzer = None  # Toulmin argument structure
10459:         self._temporal_analyzer = None  # Temporal dynamics
10460:         self._discourse_analyzer = None  # Discourse markers
10461:         self._strategic_integrator = None  # Cross-reference integration
10462:         self._causal_analyzer = None  # Causal chain analyzer (lazy-loaded)
10463:
10464:         # Modelo para clasificaciÃ³n de tipo de chunk
10465:         self.chunk_classifier = None
10466:
10467:         # Almacenamiento
10468:         self.tfidf_vectorizer = TfidfVectorizer(stop_words=self._get_stopwords(), ngram_range=(1, 2), max_df=0.85, min_df=2)
10469:         self.chunks_for_tfidf = []
10470:         self.corpus_embeddings = None
10471:
10472:     # ========================================================================
10473:     # SPECIALIZED COMPONENT PROPERTIES - Innovation layers (no canonical equivalent)
10474:     # ========================================================================
10475:
10476:     @property
10477:     def nlp(self):
10478:         """Lazy-load SpaCy NLP model"""
10479:         if self._nlp is None:
10480:             try:
10481:                 self.logger.info("Loading SpaCy model: es_core_news_lg")
10482:                 self._nlp = spacy.load("es_core_news_lg")
10483:             except:
10484:                 self.logger.warning("SpaCy es_core_news_lg no disponible, usando sm")
10485:                 try:
10486:                     self._nlp = spacy.load("es_core_news_sm")
10487:                 except:
10488:                     self.logger.error("NingÃºn modelo SpaCy disponible. Funcionalidad de NER limitada.")
10489:                     self._nlp = None
10490:         return self._nlp
10491:
10492:     @property
10493:     def context_preserver(self):
10494:         """
10495:         CANONICAL REPLACEMENT: Use semantic_processor for chunking.
10496:         Kept for backward compatibility but delegates to canonical component.
10497:         """
10498:         # Return a lightweight adapter that uses canonical SemanticProcessor
10499:         return self.semantic_processor
10500:
10501:     @property
10502:     def causal_analyzer(self):
10503:         """Lazy-load causal chain analyzer"""
10504:         if self._causal_analyzer is None:
10505:             self._causal_analyzer = CausalChainAnalyzer(self)
10506:         return self._causal_analyzer
```

```
10507:
10508:     @property
10509:     def kg_builder(self):
10510:         """Lazy-load knowledge graph builder"""
10511:         if self._kg_builder is None:
10512:             self._kg_builder = KnowledgeGraphBuilder(self)
10513:         return self._kg_builder
10514:
10515:     @property
10516:     def topic_modeler(self):
10517:         """Lazy-load topic modeler"""
10518:         if self._topic_modeler is None:
10519:             self._topic_modeler = TopicModeler(self)
10520:         return self._topic_modeler
10521:
10522:     @property
10523:     def argument_analyzer(self):
10524:         """Lazy-load argument analyzer"""
10525:         if self._argument_analyzer is None:
10526:             self._argument_analyzer = ArgumentAnalyzer(self)
10527:         return self._argument_analyzer
10528:
10529:     @property
10530:     def temporal_analyzer(self):
10531:         """Lazy-load temporal analyzer"""
10532:         if self._temporal_analyzer is None:
10533:             self._temporal_analyzer = TemporalAnalyzer(self)
10534:         return self._temporal_analyzer
10535:
10536:     @property
10537:     def discourse_analyzer(self):
10538:         """Lazy-load discourse analyzer"""
10539:         if self._discourse_analyzer is None:
10540:             self._discourse_analyzer = DiscourseAnalyzer(self)
10541:         return self._discourse_analyzer
10542:
10543:     @property
10544:     def strategic_integrator(self):
10545:         """Lazy-load strategic integrator"""
10546:         if self._strategic_integrator is None:
10547:             self._strategic_integrator = StrategicIntegrator(self)
10548:         return self._strategic_integrator
10549:
10550:     def detect_language(self, text: str) -> str:
10551:         """
10552:         Detect the primary language of a text document.
10553:
10554:         Inputs:
10555:             text (str): Text to analyze
10556:         Outputs:
10557:             str: ISO 639-1 language code (e.g., 'es', 'en', 'pt')
10558:         """
10559:         if not LANGDETECT_AVAILABLE:
10560:             # Default to Spanish for Colombian policy documents
10561:             return 'es'
10562:
```

```
10563:         try:
10564:             # Sample first 2000 characters for language detection
10565:             sample = safe_utf8_truncate(text, 2000)
10566:             detected_lang = detect(sample)
10567:             self.logger.info(f"Detected language: {detected_lang}")
10568:             return detected_lang
10569:         except Exception as e:
10570:             self.logger.warning(f"Language detection failed: {e}, defaulting to Spanish")
10571:             return 'es'
10572:
10573:     def select_embedding_model_for_language(self, language: str) -> None:
10574:         """
10575:         Select appropriate embedding model based on detected language.
10576:
10577:         Inputs:
10578:             language (str): ISO 639-1 language code
10579:         Outputs:
10580:             None - updates model selection
10581:         """
10582:         # For now, multilingual-e5-large handles multiple languages well
10583:         # Could be extended with language-specific models if needed
10584:         if language in ['es', 'pt', 'ca']:  # Spanish, Portuguese, Catalan
10585:             self.logger.info(f"Using multilingual model for {language} (optimal for Romance languages)")
10586:         else:
10587:             self.logger.info(f"Using multilingual model for {language}")
10588:
10589:         # Model is already multilingual, no change needed
10590:         # This method provides extension point for future language-specific optimization
10591:
10592:     # --- MÃ©todos de la clase principal (ContinuaciÃ³n de smart_policy_chunks_industrial_v3_complete_Version2.py) ---
10593:
10594:     def _get_stopwords(self) -> List[str]:
10595:         """
10596:         Get Spanish stopwords list.
10597:
10598:         Inputs:
10599:             None
10600:         Outputs:
10601:             List[str]: List of Spanish stopwords
10602:         """
10603:         # Una lista de stopwords mÃ¡s completa se usarÃ-a en producciÃ³n
10604:         return ['el', 'la', 'los', 'las', 'un', 'una', 'unos', 'unas', 'y', 'o', 'de', 'a', 'en', 'por', 'con', 'para', 'del', 'al', 'que', 'se', 'es', 'son
', 'han', 'como', 'mÃ¡s', 'pero', 'no', 'su', 'sus', 'ha', 'lo', 'e', 'u', 'ni', 'sin', 'mi', 'tu', 'si', 'cuando', 'este', 'esta', 'estos', 'estas', 'esos', 'esas
', 'aquel', 'aquella']
10605:
10606:     # ============================================================================
10607:     # CANONICAL SOTA METHODS - Replace manual implementations
10608:     # ============================================================================
10609:
10610:     def semantic_search_with_rerank(
10611:         self,
10612:         query: str,
10613:         chunks: list[dict],
10614:         pdq_filter: dict | None = None,
10615:         top_k: int = 10
10616:     ) -> list[tuple[dict, float]]:
```

```
        """
        CANONICAL SOTA: Semantic search with cross-encoder reranking.

        Replaces manual cosine_similarity ranking with SOTA reranker.

        Inputs:
            query (str): Search query
            chunks (list[dict]): Chunks to search
            pdq_filter (dict | None): Optional PDQ filter
            top_k (int): Number of results to return
        Outputs:
            list[tuple[dict, float]]: List of (chunk, score) tuples
        """
        results = self._spc_embed.semantic_search(
            query, chunks, pdq_filter=pdq_filter, use_reranking=True
        )
        return results[:top_k]

    def _attach_canonical_evidence(self, full_text: str) -> dict[str, Any]:
        """
        CANONICAL SOTA: Attach canonical PDQ/dimension evidence.

        Uses canonical policy patterns instead of ad-hoc heuristics.

        Inputs:
            full_text (str): Full document text
        Outputs:
            dict[str, Any]: Canonical evidence analysis
        """
        return self._spc_policy.analyze_text(full_text)

    def evaluate_numerical_consistency(
        self,
        chunks: list[dict],
        pdq_context: dict
    ) -> dict[str, Any]:
        """
        CANONICAL SOTA: Evaluate numerical consistency with Bayesian analysis.

        Uses canonical extractor + Bayesian analyzer for probabilistic scoring.

        Inputs:
            chunks (list[dict]): Chunks to evaluate
            pdq_context (dict): PDQ context for evaluation
        Outputs:
            dict[str, Any]: Numerical consistency evaluation
        """
        return self._spc_embed.evaluate_numerical_consistency(chunks, pdq_context)

    def _generate_embedding(self, text: str, model_type: str = "semantic") -> np.ndarray:
        """
        CANONICAL SOTA: Generate embedding using SemanticChunkingProducer.

        Replaces internal embedding with SOTA multilingual BGE-M3 model.
        model_type kept for compatibility; canonical pipeline is multilingual.

```

```
10673:         Inputs:
10674:             text (str): Input text to embed
10675:             model_type (str): Ignored, canonical uses SOTA multilingual
10676:         Outputs:
10677:             np.ndarray: Embedding vector from canonical SOTA component
10678:         """
10679:         return self._spc_sem.embed_text(text).astype(np.float32)
10680: 
10681:     def _create_smart_policy_chunk(
10682:         self,
10683:         strategic_unit: Dict,
10684:         metadata: Dict,
10685:         argument_structure: Optional[ArgumentStructure],
10686:         temporal_structure: Optional[TemporalDynamics],
10687:         discourse_structure: Optional[Dict],
10688:         global_topics: Dict,
10689:         global_kg: Dict
10690:     ) -> SmartPolicyChunk:
10691:         """Crear Smart Policy Chunk con anÃ¡lisis completo"""
10692:         text = strategic_unit.get("text", "")
10693:         document_id = metadata.get("document_id", "doc_001")
10694: 
10695:         # Generar embeddings
10696:         semantic_embedding = self._generate_embedding(text, 'semantic')
10697:         policy_embedding = self._generate_embedding(text, 'semantic')
10698:         causal_embedding = self._generate_embedding(text, 'semantic')
10699:         temporal_embedding = self._generate_embedding(text, 'semantic')
10700: 
10701:         # AnÃ¡lisis causales
10702:         causal_evidence = self._extract_comprehensive_causal_evidence(strategic_unit, global_kg)
10703: 
10704:         # Entidades polÃticas
10705:         policy_entities = self._extract_policy_entities_with_context(strategic_unit)
10706: 
10707:         # Contexto estratÃ©gico
10708:         strategic_context = self._derive_strategic_context(strategic_unit, global_topics)
10709: 
10710:         # MÃ©tricas de calidad
10711:         confidence_metrics = self._calculate_comprehensive_confidence(strategic_unit, causal_evidence, policy_entities)
10712:         coherence_score = self._calculate_coherence_score(strategic_unit)
10713:         completeness_index = self._calculate_completeness_index(strategic_unit, causal_evidence)
10714:         strategic_importance = self._assess_strategic_importance(strategic_unit, causal_evidence, policy_entities)
10715:         information_density = self._calculate_information_density(text)
10716:         actionability_score = self._assess_actionability(text, policy_entities)
10717: 
10718:         # Referencias cruzadas
10719:         cross_refs = self._find_cross_document_references(strategic_unit)
10720: 
10721:         # Supuestos implÃcitos
10722:         implicit_assumptions = self._extract_implicit_assumptions(strategic_unit, causal_evidence)
10723: 
10724:         # Presuposiciones contextuales
10725:         contextual_presuppositions = self._identify_contextual_presuppositions(strategic_unit)
10726: 
10727:         # Marcadores de discurso
10728:         discourse_markers = self._extract_discourse_markers(text)
```

```
10729:
10730:         # Patrones retÃ³ricos
10731:         rhetorical_patterns = self._identify_rhetorical_patterns(text)
10732:
10733:         # DistribuciÃ³n de tÃ³picos para este chunk
10734:         topic_distribution = self._calculate_chunk_topic_distribution(text, global_topics)
10735:
10736:         # Frases clave
10737:         key_phrases = self._extract_key_phrases(text)
10738:
10739:         # Nodos y aristas del grafo de conocimiento
10740:         kg_nodes = [e.normalized_form for e in policy_entities]
10741:         kg_edges = self._derive_kg_edges_for_chunk(policy_entities, causal_evidence)
10742:
10743:         # Hash y ID
10744:         content_hash = hashlib.sha256(text.encode('utf-8')).hexdigest()
10745:         chunk_id = f"{document_id}_{content_hash[:8]}"
10746:
10747:         # NormalizaciÃ³n
10748:         normalized_text = self._advanced_preprocessing(text)
10749:
10750:         return SmartPolicyChunk(
10751:             chunk_id=chunk_id,
10752:             document_id=document_id,
10753:             content_hash=content_hash,
10754:             policy_area_id=strategic_unit.get("policy_area_id"),  # PA01-PA10
10755:             dimension_id=strategic_unit.get("dimension_id"),      # DIM01-DIM06
10756:
10757:             text=text,
10758:             normalized_text=normalized_text,
10759:             semantic_density=self._calculate_semantic_density(text),
10760:
10761:             section_hierarchy=strategic_unit.get("hierarchy", []),
10762:             document_position=strategic_unit.get("position", (0, 0)),
10763:             chunk_type=self._classify_chunk_type(text),
10764:
10765:             causal_chain=causal_evidence,
10766:             policy_entities=policy_entities,
10767:             implicit_assumptions=implicit_assumptions,
10768:             contextual_presuppositions=contextual_presuppositions,
10769:
10770:             argument_structure=argument_structure,
10771:             temporal_dynamics=temporal_structure,
10772:             discourse_markers=discourse_markers,
10773:             rhetorical_patterns=rhetorical_patterns,
10774:
10775:             cross_references=cross_refs,
10776:             strategic_context=strategic_context,
10777:
10778:             confidence_metrics=confidence_metrics,
10779:             coherence_score=coherence_score,
10780:             completeness_index=completeness_index,
10781:             strategic_importance=strategic_importance,
10782:             information_density=information_density,
10783:             actionability_score=actionability_score,
10784:
```

```
10785:             semantic_embedding=semantic_embedding,
10786:             policy_embedding=policy_embedding,
10787:             causal_embedding=causal_embedding,
10788:             temporal_embedding=temporal_embedding,
10789:
10790:             knowledge_graph_nodes=kg_nodes,
10791:             knowledge_graph_edges=kg_edges,
10792:
10793:             topic_distribution=topic_distribution,
10794:             key_phrases=key_phrases,
10795:
10796:             model_versions=self._get_model_versions()
10797:         )
10798:
10799:     # --- MÃ©todos de AnÃ¡lisis y ExtracciÃ³n (ContinuaciÃ³n de smart_policy_chunks_industrial_v3_complete_Version2.py) ---
10800:
10801:     def _extract_comprehensive_causal_evidence(self, strategic_unit: Dict, global_kg: Dict) -> List[CausalEvidence]:
10802:         """Extraer evidencia causal completa (Placeholder con estructura avanzada)"""
10803:         evidence_list = []
10804:         text = strategic_unit.get("text", "")
10805:
10806:         # Placeholder para un modelo de extracciÃ³n causal mÃ¡s avanzado
10807:         # SimulaciÃ³n de extracciÃ³n basada en patrones de palabras
10808:
10809:         dimensions = {
10810:             'problem_solution': [r'soluciÃ³n\s+para\s+([^.]+)', r'abordar\s+([^.]+)\s+mediante\s+([^.]+)'],
10811:             'impact_assessment': [r'tendrÃ¡\s+un\s+impacto\s+([^.]+)', r'conducirÃ¡\s+a\s+([^.]+)'],
10812:             'resource_allocation': [r'asignaciÃ³n\s+de\s+([^,]+)\s+para\s+([^.]+)'],
10813:             'policy_instrument': [r'(?:la\s+implementaciÃ³n|el\s+uso)\s+de\s+([^,]+)\s+resultarÃ¡\s+en\s+([^.]+)']
10814:         }
10815:
10816:         for dimension, patterns in dimensions.items():
10817:             for pattern in patterns:
10818:                 matches = list(re.finditer(pattern, text, re.IGNORECASE))
10819:                 for match in matches:
10820:                     context_start = max(0, match.start() - 150)
10821:                     context_end = min(len(text), match.end() + 150)
10822:                     context = text[context_start:context_end]
10823:
10824:                     # Determinar tipo de relaciÃ³n causal
10825:                     causal_type = self._determine_causal_type(match.group(0), context)
10826:
10827:                     # Calcular fuerza de la relaciÃ³n
10828:                     strength = self._calculate_causal_strength(match.group(0), context)
10829:
10830:                     # Identificar mecanismos
10831:                     mechanisms = self._identify_causal_mechanisms(context)
10832:
10833:                     evidence_list.append(CausalEvidence(
10834:                         dimension=dimension,
10835:                         category=self._categorize_causal_evidence(dimension),
10836:                         matches=[match.group(0)],
10837:                         confidence=0.75 + (strength * 0.2), # Ajuste por fuerza
10838:                         context_span=(context_start, context_end),
10839:                         implicit_indicators=self._find_implicit_indicators(context),
10840:                         causal_type=causal_type,
```

```
                        strength_score=strength,
                        mechanisms=mechanisms,
                        confounders=self._identify_confounders(context),
                        mediators=self._identify_mediators(context),
                        moderators=self._identify_moderators(context)
                    ))

        return evidence_list

    def _categorize_causal_evidence(self, dimension: str) -> str:
        """Categorizar el tipo de evidencia causal"""
        if dimension in ['problem_solution', 'impact_assessment']:
            return 'macro_policy'
        elif dimension in ['resource_allocation', 'policy_instrument']:
            return 'implementation_mechanisms'
        return 'general'

    def _determine_causal_type(self, match_text: str, context: str) -> CausalRelationType:
        """Determinar el tipo de relaciÃ³n causal por marcadores lingÃ¼Ãsticos"""
        match_lower = match_text.lower()
        if 'si' in match_lower and 'entonces' in context.lower():
            return CausalRelationType.CONDITIONAL
        elif any(word in match_lower for word in ['porque', 'debido a', 'gracias a']):
            return CausalRelationType.DIRECT_CAUSE
        elif any(word in match_lower for word in ['por lo tanto', 'en consecuencia']):
            return CausalRelationType.DIRECT_CAUSE
        else:
            return CausalRelationType.INDIRECT_CAUSE

    def _calculate_causal_strength(self, match_text: str, context: str) -> float:
        """Calcular fuerza de relaciÃ³n causal"""
        strength_score = 0.5

        # Reforzadores (Boosters)
        boosters = ['directamente', 'significativamente', 'claramente', 'contundentemente']
        if any(b in context.lower() for b in boosters):
            strength_score += 0.3

        # Mitigadores (Dampeners)
        dampeners = ['parcialmente', 'limitadamente', 'posiblemente', 'podrÃ-a']
        if any(d in context.lower() for d in dampeners):
            strength_score -= 0.3

        return np.clip(strength_score, 0.1, 1.0)

    def _identify_causal_mechanisms(self, context: str) -> List[str]:
        """Identificar mecanismos causales (cÃ³mo funciona la relaciÃ³n)"""
        mechanisms = []
        mechanism_patterns = [
            r'a\s+travÃ©s\s+de\s+([^.]+)',
            r'mediante\s+([^.]+)'
        ]

        for pattern in mechanism_patterns:
            matches = re.findall(pattern, context, re.IGNORECASE)
            mechanisms.extend(matches[:2])
```

```
10897:
10898:         return mechanisms
10899:
10900:     def _find_implicit_indicators(self, context: str) -> List[str]:
10901:         """Encontrar indicadores implÃcitos de causalidad (sin marcadores explÃcitos)"""
10902:         implicit = []
10903:         implicit_patterns = [
10904:             r'para\s+lograr\s+([^.]+)',
10905:             r'es\s+fundamental\s+([^.]+)',
10906:             r'la\s+falta\s+de\s+([^.]+)\s+impacta\s+([^.]+)'
10907:         ]
10908:
10909:         for pattern in implicit_patterns:
10910:             matches = re.findall(pattern, context, re.IGNORECASE)
10911:             implicit.extend(matches[:2])
10912:
10913:         return implicit
10914:
10915:     def _identify_confounders(self, context: str) -> List[str]:
10916:         """Identificar factores de confusiÃ³n causales"""
10917:         confounders = []
10918:         confounder_patterns = [
10919:             r'a\s+pesar\s+de\s+([^,]+)',
10920:             r'sin\s+considerar\s+([^,]+)'
10921:         ]
10922:
10923:         for pattern in confounder_patterns:
10924:             matches = re.findall(pattern, context, re.IGNORECASE)
10925:             confounders.extend(matches[:2])
10926:
10927:         return confounders
10928:
10929:     def _identify_mediators(self, context: str) -> List[str]:
10930:         """Identificar mediadores causales"""
10931:         mediators = []
10932:         mediator_patterns = [
10933:             r'que\s+a\s+su\s+vez\s+([^.]+)',
10934:             r'lo\s+cual\s+([^.]+)'
10935:         ]
10936:
10937:         for pattern in mediator_patterns:
10938:             matches = re.findall(pattern, context, re.IGNORECASE)
10939:             mediators.extend(matches[:2])
10940:
10941:         return mediators
10942:
10943:     def _identify_moderators(self, context: str) -> List[str]:
10944:         """Identificar moderadores causales"""
10945:         moderators = []
10946:         moderator_patterns = [
10947:             r'en\s+la\s+medida\s+(?:en\s+)?que\s+([^,]+)',
10948:             r'siempre\s+y\s+cuando\s+([^,]+)',
10949:             r'dependiendo\s+de\s+([^,]+)'
10950:         ]
10951:
10952:         for pattern in moderator_patterns:
```

```
            matches = re.findall(pattern, context, re.IGNORECASE)
            moderators.extend(matches[:2])

        return moderators

    def _extract_policy_entities_with_context(self, strategic_unit: Dict) -> List[PolicyEntity]:
        """Extraer entidades de polÃtica con contexto completo"""
        entities = []
        text = strategic_unit.get("text", "")

        # Patrones de entidades institucionales
        institutional_patterns = [
            (r'(?:Ministerio|SecretarÃa|Departamento|DirecciÃ³n|Instituto|Agencia)\s+(?:de|del?|para)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©ÃÃ³Ã±\s]+', 'institution'),
            (r'(?:AlcaldÃa|GobernaciÃ³n|Prefectura)\s+(?:de|del?)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©ÃÃ³Ã±\s]+', 'local_government'),
            (r'(?:Consejo|ComitÃ©|Junta)\s+(?:de|del?|para)\s+[A-ZÃ\201Ã\211Ã\215Ã\223Ã\232Ã\221][a-zÃ¡Ã©ÃÃ³Ã±\s]+', 'committee')
        ]

        for pattern, entity_type in institutional_patterns:
            matches = re.finditer(pattern, text)
            for match in matches:
                role = self._infer_entity_role(match.group(0), text)
                entities.append(PolicyEntity(
                    entity_type=entity_type,
                    text=match.group(0),
                    normalized_form=self._normalize_entity(match.group(0)),
                    context_role=role,
                    confidence=0.7,
                    span=match.span()
                ))

        # Patrones de poblaciÃ³n/beneficiarios (simplificado)
        beneficiary_patterns = [
            (r'(?:los|las)\s+(?:niÃ±os|niÃ±as|jÃ³venes|mujeres|poblaciÃ³n|comunidad|ciudadanos)\s+(?:de|en)\s+[^,.]+', 'population_group'),
            (r'beneficiarios\s+(?:de|del?)\s+[^,.]+', 'beneficiary_group')
        ]

        for pattern, entity_type in beneficiary_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                entities.append(PolicyEntity(
                    entity_type=entity_type,
                    text=match.group(0),
                    normalized_form=self._normalize_entity(match.group(0)),
                    context_role=PolicyEntityRole.BENEFICIARY,
                    confidence=0.8,
                    span=match.span()
                ))

        # DeduplicaciÃ³n y conteo
        unique_entities: List[PolicyEntity] = []
        seen_texts = set()
        for entity in entities:
            normalized = entity.normalized_form.lower()
            if normalized not in seen_texts:
                seen_texts.add(normalized)
```

```
                unique_entities.append(entity)
            else:
                # Actualizar contador de menciones
                for existing in unique_entities:
                    if existing.normalized_form.lower() == normalized:
                        existing.mentioned_count += 1
                        break

        return unique_entities

    def _infer_entity_role(self, entity_text: str, context: str) -> PolicyEntityRole:
        """Inferir el rol de la entidad en el contexto"""
        context_lower = context.lower()
        if any(term in context_lower for term in [f"ejecutarÃ¡ {entity_text.lower()}", f"responsable de {entity_text.lower()}", f"{entity_text.lower()} impl
ementarÃ¡"]):
            return PolicyEntityRole.EXECUTOR
        elif any(term in context_lower for term in [f"beneficiarÃ¡ a {entity_text.lower()}", f"dirigido a {entity_text.lower()}"]):
            return PolicyEntityRole.BENEFICIARY
        elif any(term in context_lower for term in [f"regulador {entity_text.lower()}", f"{entity_text.lower()} emitirÃ¡"]):
            return PolicyEntityRole.REGULATOR
        elif any(term in context_lower for term in [f"financiarÃ¡ {entity_text.lower()}", f"funder {entity_text.lower()}"]):
            return PolicyEntityRole.FUNDER
        else:
            return PolicyEntityRole.STAKEHOLDER

    def _normalize_entity(self, entity_text: str) -> str:
        """Normalizar la forma de la entidad"""
        # Eliminar artÃ-culos, preposiciones y estandarizar mayÃºsculas
        text = re.sub(r'\b(el|la|los|las|un|una|de|del|a|en|por)\b', '', entity_text, flags=re.IGNORECASE).strip()
        text = re.sub(r'\s+', ' ', text)
        return text.title()

    def _derive_strategic_context(self, strategic_unit: Dict, global_topics: Dict) -> StrategicContext:
        """Derivar contexto estratÃ©gico comprehensivo"""
        text = strategic_unit.get("text", "")

        return StrategicContext(
            policy_intent=self._infer_policy_intent(text),
            implementation_phase=self._identify_implementation_phase(text),
            geographic_scope=self._extract_geographic_scope(text),
            temporal_horizon=self._determine_temporal_horizon(text),
            budget_linkage=self._identify_budget_linkage(text),
            risk_factors=self._extract_risk_factors(text),
            success_indicators=self._identify_success_indicators(text),
            alignment_with_sdg=self._identify_sdg_alignment(text),
            stakeholder_map=self._build_stakeholder_map(text),
            policy_coherence_score=self._calculate_policy_coherence(text, global_topics),
            intervention_logic_chain=self._extract_intervention_logic(strategic_unit)
        )

    def _infer_policy_intent(self, text: str) -> str:
        """Inferir la intenciÃ³n de polÃ-tica (e.g., mitigar, promover, regular)"""
        text_lower = text.lower()
        if 'promover' in text_lower or 'fomentar' in text_lower or 'impulsar' in text_lower:
            return 'PromociÃ³n/Fomento'
        elif 'reducir' in text_lower or 'mitigar' in text_lower or 'combatir' in text_lower:
```

```
11063:             return 'MitigaciÃ³n/ReducciÃ³n'
11064:         elif 'regular' in text_lower or 'establecer normativa' in text_lower or 'ley' in text_lower:
11065:             return 'RegulaciÃ³n/Normativa'
11066:         return 'General'
11067:
11068:     def _identify_implementation_phase(self, text: str) -> str:
11069:         """Identificar la fase de implementaciÃ³n (e.g., diseÃ±o, ejecuciÃ³n, evaluaciÃ³n)"""
11070:         text_lower = text.lower()
11071:         if 'diseÃ±o' in text_lower or 'formulaciÃ³n' in text_lower:
11072:             return 'DiseÃ±o'
11073:         elif 'ejecuciÃ³n' in text_lower or 'implementaciÃ³n' in text_lower or 'puesta en marcha' in text_lower:
11074:             return 'EjecuciÃ³n'
11075:         elif 'evaluaciÃ³n' in text_lower or 'seguimiento' in text_lower or 'monitoreo' in text_lower:
11076:             return 'EvaluaciÃ³n'
11077:         return 'Mixto'
11078:
11079:     def _extract_geographic_scope(self, text: str) -> str:
11080:         """Extraer el alcance geogrÃ¡fico"""
11081:         text_lower = text.lower()
11082:         if 'nacional' in text_lower or 'paÃ-s' in text_lower:
11083:             return 'Nacional'
11084:         elif 'departamental' in text_lower or 'departamento' in text_lower or 'provincia' in text_lower:
11085:             return 'Departamental'
11086:         elif 'municipal' in text_lower or 'municipio' in text_lower or 'ciudad' in text_lower:
11087:             return 'Municipal/Local'
11088:         return 'Indefinido'
11089:
11090:     def _determine_temporal_horizon(self, text: str) -> str:
11091:         """Determinar el horizonte temporal (corto, mediano, largo plazo)"""
11092:         text_lower = text.lower()
11093:         if 'corto plazo' in text_lower or 'prÃ³ximo aÃ±o' in text_lower:
11094:             return 'Corto Plazo'
11095:         elif 'mediano plazo' in text_lower or 'prÃ³ximos 4 aÃ±os' in text_lower:
11096:             return 'Mediano Plazo'
11097:         elif 'largo plazo' in text_lower or 'horizonte 2030' in text_lower:
11098:             return 'Largo Plazo'
11099:         return 'Mixto'
11100:
11101:     def _identify_budget_linkage(self, text: str) -> str:
11102:         """Identificar vÃ-nculos presupuestales"""
11103:         text_lower = text.lower()
11104:         if 'presupuesto' in text_lower or 'recursos financieros' in text_lower or 'inversiÃ³n de' in text_lower:
11105:             return 'ExplÃ-cito'
11106:         elif 'costos' in text_lower or 'financiamiento' in text_lower:
11107:             return 'ImplÃ-cito'
11108:         return 'Ausente'
11109:
11110:     def _extract_risk_factors(self, text: str) -> List[str]:
11111:         """Extraer factores de riesgo explÃ-citos"""
11112:         risks = []
11113:         risk_patterns = [
11114:             r'el\s+riesgo\s+(?:de|es)\s+([^.]+)',
11115:             r'se\s+deben\s+mitigar\s+([^.]+)'
11116:         ]
11117:         for pattern in risk_patterns:
11118:             matches = re.findall(pattern, text, re.IGNORECASE)
```

```
            risks.extend(matches[:3])
        return risks

    def _identify_success_indicators(self, text: str) -> List[str]:
        """Identificar indicadores de Ã©xito o mÃ©tricas"""
        indicators = []
        indicator_patterns = [
            r'indicador\s+([^.]+)',
            r'meta\s+(?:de|para)\s+([^.]+)',
            r'se\s+medirÃ¡\s+con\s+([^.]+)'
        ]
        for pattern in indicator_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            indicators.extend(matches[:3])
        return indicators

    def _identify_sdg_alignment(self, text: str) -> List[str]:
        """Identificar alineaciÃ³n con ODS"""
        sdgs = []
        # ODS explÃcitos
        sdg_pattern = r'ODS\s+(\d+)'
        matches = re.findall(sdg_pattern, text, re.IGNORECASE)
        sdgs.extend([f"ODS_{m}" for m in matches])

        # ODS por temas (simplificado)
        sdg_themes = {
            'ODS_1': ['pobreza', 'pobres'],
            'ODS_2': ['hambre', 'alimentaciÃ³n', 'nutriciÃ³n'],
            'ODS_3': ['salud', 'bienestar'],
            'ODS_4': ['educaciÃ³n', 'calidad educativa'],
            'ODS_5': ['igualdad de gÃ©nero', 'mujeres'],
            'ODS_6': ['agua', 'saneamiento'],
            'ODS_7': ['energÃa'],
            'ODS_8': ['empleo', 'crecimiento econÃ³mico'],
            'ODS_10': ['desigualdad', 'equidad'],
            'ODS_11': ['ciudades sostenibles', 'desarrollo urbano'],
            'ODS_13': ['cambio climÃ¡tico', 'clima'],
            'ODS_16': ['paz', 'justicia', 'instituciones']
        }
        text_lower = text.lower()
        for sdg, themes in sdg_themes.items():
            if any(theme in text_lower for theme in themes) and sdg not in sdgs:
                sdgs.append(sdg)

        return sorted(list(set(sdgs)))

    def _build_stakeholder_map(self, text: str) -> Dict[str, List[str]]:
        """Construir mapa de stakeholders (simplificado)"""
        stakeholders = defaultdict(list)
        # Reutilizar inferencia de roles para llenar el mapa
        entities = self._extract_policy_entities_with_context({'text': text})
        for entity in entities:
            role_name = entity.context_role.name.lower()
            if entity.normalized_form not in stakeholders[role_name]:
                stakeholders[role_name].append(entity.normalized_form)
        return dict(stakeholders)
```

```
11175:
11176:     def _calculate_policy_coherence(self, text: str, global_topics: Dict) -> float:
11177:         """Calcular coherencia de polÃtica (alineaciÃ³n con temas clave)"""
11178:         # SimplificaciÃ³n: media de la alineaciÃ³n tÃ³pica
11179:         return self._calculate_topic_alignment(text, global_topics)
11180:
11181:     def _extract_intervention_logic(self, strategic_unit: Dict) -> List[str]:
11182:         """Extraer la cadena lÃ³gica de intervenciÃ³n (e.g., insumo -> actividad -> producto -> resultado)"""
11183:         logic = []
11184:
11185:         # SimplificaciÃ³n: encadenamiento de antecedentes y consecuentes
11186:         for chain in strategic_unit.get('chains', []):
11187:             if chain.get('antecedent') and chain.get('consequent'):
11188:                 logic.append(f"{chain['antecedent']} -> {chain['consequent']}")
11189:
11190:         return logic[:5]
11191:
11192:     def _calculate_comprehensive_confidence(
11193:         self,
11194:         strategic_unit: Dict,
11195:         causal_evidence: List[CausalEvidence],
11196:         policy_entities: List[PolicyEntity]
11197:     ) -> Dict[str, float]:
11198:         """Calcular mÃ©tricas de confianza y calidad"""
11199:
11200:         weights = {
11201:             'causal': 0.4,
11202:             'entity': 0.3,
11203:             'structural': 0.2,
11204:             'semantic': 0.1
11205:         }
11206:
11207:         causal_conf = np.mean([e.confidence for e in causal_evidence]) if causal_evidence else 0.0
11208:         entity_conf = np.mean([e.confidence for e in policy_entities]) if policy_entities else 0.0
11209:         structural_conf = strategic_unit.get('confidence', 0.7)
11210:         semantic_conf = strategic_unit.get('semantic_coherence', 0.6)
11211:
11212:         overall = (
11213:             weights['causal'] * causal_conf +
11214:             weights['entity'] * entity_conf +
11215:             weights['structural'] * structural_conf +
11216:             weights['semantic'] * semantic_conf
11217:         )
11218:
11219:         return {
11220:             'causal_confidence': causal_conf,
11221:             'entity_confidence': entity_conf,
11222:             'structural_confidence': structural_conf,
11223:             'semantic_confidence': semantic_conf,
11224:             'overall_confidence': min(overall, 1.0)
11225:         }
11226:
11227:     def _find_cross_document_references(self, strategic_unit: Dict) -> List[CrossDocumentReference]:
11228:         """Encontrar referencias cruzadas documentales (Placeholder)"""
11229:         references = []
11230:         text = strategic_unit.get("text", "")
```

```
        ref_patterns = [
            (r'(?:ver|vÃ©ase)\s+(?:secciÃ³n|capÃ-tulo)\s+([^,\.]+)', 'explicit_reference'),
            (r'como\s+se\s+(?:mencionÃ³|indicÃ³)\s+en\s+([^,\.]+)', 'backward_reference'),
            (r'se\s+desarrollarÃ¡\s+en\s+([^,\.]+)', 'forward_reference')
        ]

        for pattern, ref_type in ref_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                references.append(CrossDocumentReference(
                    target_section=match.group(1).strip(),
                    reference_type=ref_type,
                    confidence=0.7,
                    semantic_linkage=0.6,
                    context_bridge=match.group(0)
                ))

        return references

    def _extract_implicit_assumptions(self, strategic_unit: Dict, causal_evidence: List[CausalEvidence]) -> List[Tuple[str, float]]:
        """Extraer supuestos implÃ-citos (Placeholder)"""
        assumptions = []
        # Los antecedentes condicionales de baja confianza se consideran supuestos
        for chain in strategic_unit.get('chains', []):
            if chain.get('type') == 'conditional' and chain.get('confidence', 0.5) < 0.6:
                assumptions.append((f"Asumiendo que: {chain['antecedent']}", 1.0 - chain.get('confidence', 0.5)))
        return assumptions

    def _identify_contextual_presuppositions(self, strategic_unit: Dict) -> List[Tuple[str, float]]:
        """Identificar presuposiciones contextuales (Placeholder)"""
        presuppositions = []
        text_lower = strategic_unit.get("text", "").lower()

        if 'se continuarÃ¡' in text_lower or 'marco existente' in text_lower:
            presuppositions.append(("Existe un marco de polÃ-tica previo", 0.8))
        if 'presupuesto asignado' in text_lower:
            presuppositions.append(("Se cuenta con la disponibilidad de recursos", 0.9))

        return presuppositions

    def _extract_discourse_markers(self, text: str) -> List[Tuple[str, str]]:
        """Extraer marcadores de discurso (conectores)"""
        markers = []

        coherence_markers = {
            'addition': [r'\by\b', r'\btambiÃ©n\b', r'\badicionalmente\b'],
            'contrast': [r'\bpero\b', r'\bsin\s+embargo\b', r'\bno\s+obstante\b'],
            'cause': [r'\bporque\b', r'\bya\s+que\b', r'\bdebido\s+a\b'],
            'result': [r'\bpor\s+lo\s+tanto\b', r'\bpor\s+ende\b', r'\ben\s+consecuencia\b']
        }

        for relation_type, patterns in coherence_markers.items():
            for pattern in patterns:
                matches = re.finditer(pattern, text, re.IGNORECASE)
                for match in matches:
                    markers.append((match.group(0), relation_type))
```

```
11287:
11288:         return markers
11289:
11290:     def _identify_rhetorical_patterns(self, text: str) -> List[str]:
11291:         """Identificar patrones retÃ³ricos (e.g., afirmaciÃ³n, evidencia, conclusiÃ³n)"""
11292:         rhetorical = []
11293:         rhetorical_patterns = self._analyze_rhetorical_structure(text)
11294:
11295:         for key, value in rhetorical_patterns.items():
11296:             if value:
11297:                 rhetorical.append(key)
11298:
11299:         return rhetorical
11300:
11301:     def _calculate_chunk_topic_distribution(self, text: str, global_topics: Dict) -> Dict[str, float]:
11302:         """Calcular distribuciÃ³n tÃ³pica del chunk"""
11303:         distribution = {}
11304:         text_lower = text.lower()
11305:
11306:         for topic in global_topics.get('topics', []):
11307:             topic_score = 0
11308:             for keyword, _ in topic['keywords'][:10]:
11309:                 if keyword.lower() in text_lower:
11310:                     topic_score += 1
11311:             if topic_score > 0:
11312:                 distribution[f"topic_{topic['topic_id']}"] = topic_score / 10.0
11313:
11314:         return distribution
11315:
11316:     def _extract_key_phrases(self, text: str) -> List[Tuple[str, float]]:
11317:         """Extraer frases clave del texto (Placeholder: usando TF-IDF)"""
11318:         try:
11319:             # Usar TF-IDF para frases clave
11320:             if not self.chunks_for_tfidf:
11321:                 return []
11322:
11323:             tfidf_matrix = self.tfidf_vectorizer.fit_transform(self.chunks_for_tfidf)
11324:             feature_names = self.tfidf_vectorizer.get_feature_names_out()
11325:
11326:             # Vectorizar solo el texto actual
11327:             text_vector = self.tfidf_vectorizer.transform([text])
11328:
11329:             # Obtener los top features para este documento
11330:             feature_array = text_vector.toarray().flatten()
11331:             top_indices = feature_array.argsort()[-10:][::-1]
11332:
11333:             key_phrases = [(feature_names[i], feature_array[i]) for i in top_indices if feature_array[i] > 0]
11334:             return key_phrases
11335:
11336:         except Exception as e:
11337:             self.logger.error(f"Error en extracciÃ³n de frases clave: {e}")
11338:             return []
11339:
11340:     def _derive_kg_edges_for_chunk(self, entities: List[PolicyEntity], causal_evidence: List[CausalEvidence]) -> List[Tuple[str, str, str, float]]:
11341:         """Derivar aristas del grafo de conocimiento para el chunk"""
11342:         edges = []
```

```
11343:
11344:         # Entidades y relaciones causales
11345:         for entity1 in entities:
11346:             for entity2 in entities:
11347:                 if entity1 != entity2:
11348:                     for evidence in causal_evidence:
11349:                         # Simple heurÃ-stica: si ambas entidades estÃ¡n en el contexto causal
11350:                         context = evidence.context_span
11351:                         if entity1.span[0] >= context[0] and entity2.span[1] <= context[1]:
11352:                             edges.append((
11353:                                 entity1.normalized_form,
11354:                                 entity2.normalized_form,
11355:                                 evidence.causal_type.value,
11356:                                 evidence.confidence
11357:                             ))
11358:
11359:         return edges[:10]
11360:
11361:     def _get_model_versions(self) -> Dict[str, str]:
11362:         """Devuelve las versiones de los modelos utilizados"""
11363:         return {
11364:             'semantic_model': 'intfloat/multilingual-e5-large',
11365:             'spacy_model': self.nlp.meta.get('name') if self.nlp else 'None',
11366:             'pipeline_version': 'SMART-CHUNK-3.0-FINAL'
11367:         }
11368:
11369:     def _calculate_semantic_density(self, text: str) -> float:
11370:         """Calcular la densidad semÃ¡ntica del chunk (placeholder)"""
11371:         # Densidad basada en la proporciÃ³n de palabras clave
11372:         return min(len(re.findall(r'\b[A-Z][a-z]+\b', text)) / len(text.split()), 1.0)
11373:
11374:     def _classify_chunk_type(self, text: str) -> ChunkType:
11375:         """Clasificar el tipo de chunk basado en el contenido (Placeholder)"""
11376:         text_lower = text.lower()
11377:         type_indicators = {
11378:             ChunkType.DIAGNOSTICO: ['diagnÃ³stico', 'anÃ¡lisis', 'situaciÃ³n actual'],
11379:             ChunkType.ESTRATEGIA: ['estrategia', 'objetivo', 'meta', 'propÃ³sito'],
11380:             ChunkType.METRICA: ['indicador', 'meta', 'mediciÃ³n', 'lÃ-nea base'],
11381:             ChunkType.FINANCIERO: ['presupuesto', 'recursos financieros', 'inversiÃ³n'],
11382:             ChunkType.NORMATIVO: ['ley', 'decreto', 'normativa', 'regulaciÃ³n'],
11383:             ChunkType.OPERATIVO: ['operaciÃ³n', 'implementaciÃ³n', 'ejecuciÃ³n'],
11384:             ChunkType.EVALUACION: ['evaluaciÃ³n', 'seguimiento', 'monitoreo']
11385:         }
11386:
11387:         scores = {}
11388:         for chunk_type, indicators in type_indicators.items():
11389:             score = sum(1 for ind in indicators if ind in text_lower)
11390:             if score > 0:
11391:                 scores[chunk_type] = score
11392:
11393:         if scores:
11394:             return max(scores, key=scores.get)
11395:         return ChunkType.MIXTO
11396:
11397:     def _calculate_coherence_score(self, strategic_unit: Dict) -> float:
11398:         """Calcular puntuaciÃ³n de coherencia"""
```

```
11399:         return strategic_unit.get('coherence', 0.75)
11400:
11401:     def _calculate_completeness_index(
11402:         self,
11403:         strategic_unit: Dict,
11404:         causal_evidence: List[CausalEvidence]
11405:     ) -> float:
11406:         """Calcular Ã-ndice de completitud"""
11407:         components = {
11408:             'has_text': bool(strategic_unit.get('text')),
11409:             'has_causal': len(causal_evidence) > 0,
11410:             'has_entities': len(strategic_unit.get('entities', [])) > 0,
11411:             'has_context': bool(strategic_unit.get('context')),
11412:             'has_hierarchy': bool(strategic_unit.get('hierarchy')),
11413:             'has_arg_structure': bool(strategic_unit.get('argument_structure')),
11414:             'has_temporal': bool(strategic_unit.get('temporal_dynamics'))
11415:         }
11416:
11417:         return sum(components.values()) / len(components)
11418:
11419:     # --- MÃ©todos de la clase principal (ContinuaciÃ³n de smart_policy_chunks_industrial_v3_complete_final_Version2.py) ---
11420:
11421:     def _analyze_cross_references(self, text: str) -> Dict[str, List[Dict[str, Any]]]:
11422:         """AnÃ¡lisis de referencias cruzadas y relaciones inter-documento"""
11423:         references = defaultdict(list)
11424:
11425:         reference_patterns = [
11426:             (r'como\s+se\s+(?:mencionÃ³|indicÃ³|seÃ±alÃ³|estableciÃ³)\s+(?:en\s+)?(.+?)[,.]', 'backward'),
11427:             (r'(?:ver|vÃ©ase|consultar)\s+(?:la\s+)?(?:secciÃ³n|capÃ-tulo|apartado)\s+(.+?)[,.]', 'explicit'),
11428:             (r'tal\s+como\s+se\s+establece\s+en\s+(.+?)[,.]', 'normative'),
11429:             (r'de\s+acuerdo\s+(?:con|a)\s+(?:lo\s+establecido\s+en\s+)?(.+?)[,.]', 'normative'),
11430:             (r'segÃºn\s+(?:lo\s+dispuesto\s+en\s+)?(.+?)[,.]', 'normative'),
11431:             (r'conforme\s+a\s+(.+?)[,.]', 'normative'),
11432:             (r'en\s+el\s+marco\s+de\s+(.+?)[,.]', 'framework'),
11433:             (r'se\s+desarrollarÃ¡\s+en\s+(.+?)[,.]', 'forward')
11434:         ]
11435:
11436:         for pattern, ref_type in reference_patterns:
11437:             matches = re.finditer(pattern, text, re.IGNORECASE)
11438:             for match in matches:
11439:                 references[ref_type].append({
11440:                     'target': match.group(1).strip(),
11441:                     'position': match.span(),
11442:                     'context': text[max(0, match.start()-100):min(len(text), match.end()+100)]
11443:                 })
11444:
11445:         return dict(references)
11446:
11447:     def _analyze_temporal_structure(self, text: str) -> Dict[str, Any]:
11448:         """AnÃ¡lisis temporal completo"""
11449:         temporal_info = {
11450:             'time_markers': [],
11451:             'sequences': [],
11452:             'durations': [],
11453:             'milestones': [],
11454:             'temporal_ordering': []
```

```
        }

        # Marcadores temporales explÃ-citos
        time_patterns = [
            (r'\b(20\d{2})\b', 'year'),
            (r'\b(enero|febrero|marzo|abril|mayo|junio|julio|agosto|septiembre|octubre|noviembre|diciembre)\s+(?:de\s+)?(20\d{2})\b', 'month_year'),
            (r'\b(?:en|durante|hasta|desde|para)\s+(20\d{2})\b', 'temporal_prep'),
            (r'\b(corto|mediano|largo)\s+plazo\b', 'horizon'),
            (r'\b(trimestre|semestre|bimestre|cuatrimestre)\s+(\d+)\b', 'period'),
            (r'\b(?:primer|segundo|tercer|cuarto)\s+(?:trimestre|semestre)\b', 'period_ordinal'),
            (r'\b(inmediato|urgente|prioritario)\b', 'urgency'),
            (r'\b(?:antes\s+de|despuÃ©s\s+de|al\s+finalizar|una\s+vez\s+que)\b', 'sequence')
        ]

        for pattern, temp_type in time_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                temporal_info['time_markers'].append({
                    'text': match.group(0),
                    'type': temp_type,
                    'position': match.span()
                })

        # Secuencias temporales
        sequence_patterns = [
            r'(?:primero|en\s+primer\s+lugar)',
            r'(?:segundo|en\s+segundo\s+lugar|luego|posteriormente)',
            r'(?:tercero|en\s+tercer\s+lugar|finalmente|por\s+Ãºltimo)'
        ]

        for idx, pattern in enumerate(sequence_patterns):
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                temporal_info['sequences'].append({
                    'order': idx + 1,
                    'marker': match.group(0),
                    'position': match.start()
                })

        return temporal_info

    def _analyze_discourse_structure(self, text: str) -> Dict[str, Any]:
        """AnÃ¡lisis de estructura discursiva"""
        discourse = {
            'sentences': [],
            'paragraphs': [],
            'discourse_relations': [],
            'rhetorical_moves': []
        }

        if self.nlp:
            # Safe UTF-8 truncation for memory limit
            truncated_text = safe_utf8_truncate(text, 500000)
            doc = self.nlp(truncated_text)
            discourse['sentences'] = [sent.text for sent in doc.sents][:1000]

```

```
            # Extraer entidades nombradas
            discourse['entities'] = [(ent.text, ent.label_) for ent in doc.ents][:200]

            # Chunks nominales
            discourse['noun_chunks'] = [chunk.text for chunk in doc.noun_chunks][:200]
        else:
            # Fallback sin SpaCy
            discourse['sentences'] = re.split(r'[.!?]+\s+', text)[:1000]

        # Detectar pÃ¡rrafos
        paragraphs = text.split('\n\n')
        discourse['paragraphs'] = [p.strip() for p in paragraphs if p.strip()][:500]

        return discourse

    def _extract_coherence_relations(self, text: str) -> List[Dict[str, Any]]:
        """Extraer relaciones de coherencia discursiva"""
        relations = []

        coherence_markers = {
            'addition': [
                r'\by\b', r'\be\b', r'\btambiÃ©n\b', r'\basimismo\b',
                r'\badicionalmente\b', r'\bademÃ¡s\b', r'\bigualmente\b'
            ],
            'contrast': [
                r'\bpero\b', r'\bsin\s+embargo\b', r'\bno\s+obstante\b',
                r'\ben\s+cambio\b', r'\bpor\s+el\s+contrario\b', r'\bmientras\s+que\b'
            ],
            'cause': [
                r'\bporque\b', r'\bya\s+que\b', r'\bdebido\s+a\b',
                r'\bpuesto\s+que\b', r'\bdado\s+que\b', r'\ba\s+causa\s+de\b'
            ],
            'result': [
                r'\bpor\s+lo\s+tanto\b', r'\bpor\s+ende\b', r'\basÃ-\b',
                r'\ben\s+consecuencia\b', r'\bpor\s+consiguiente\b', r'\bde\s+modo\s+que\b'
            ],
            'condition': [
                r'\bsi\b', r'\ben\s+caso\s+de\s+que\b', r'\bsiempre\s+que\b',
                r'\bcon\s+tal\s+de\s+que\b', r'\ba\s+menos\s+que\b'
            ],
            'purpose': [
                r'\bpara\s+que\b', r'\ba\s+fin\s+de\s+que\b', r'\bcon\s+el\s+objetivo\s+de\b',
                r'\bcon\s+el\s+propÃ³sito\s+de\b'
            ],
            'elaboration': [
                r'\bes\s+decir\b', r'\bo\s+sea\b', r'\ben\s+otras\s+palabras\b',
                r'\bdicho\s+de\s+otro\s+modo\b'
            ],
            'example': [
                r'\bpor\s+ejemplo\b', r'\bcomo\s+es\s+el\s+caso\s+de\b',
                r'\btal\s+como\b', r'\bverbigracia\b'
            ]
        }

        for relation_type, markers in coherence_markers.items():
            for marker in markers:
```

```
                matches = re.finditer(marker, text, re.IGNORECASE)
                for match in matches:
                    relations.append({
                        'type': relation_type,
                        'marker': match.group(0),
                        'position': match.span(),
                        'confidence': 0.8
                    })

        return relations

    def _analyze_rhetorical_structure(self, text: str) -> Dict[str, List[str]]:
        """Analizar estructura retÃ³rica del texto"""
        rhetorical = {
            'claims': [],
            'evidence': [],
            'examples': [],
            'conclusions': [],
            'justifications': []
        }

        # Patrones de claims (afirmaciones)
        claim_patterns = [
            r'se\s+propone\s+que\s+[^.]+',
            r'es\s+necesario\s+[^.]+',
            r'se\s+debe\s+[^.]+',
            r'resulta\s+fundamental\s+[^.]+',
            r'se\s+requiere\s+[^.]+'
        ]

        for pattern in claim_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            rhetorical['claims'].extend(matches[:50])

        # Patrones de evidencia
        evidence_patterns = [
            r'segÃºn\s+[^,]+,',
            r'de\s+acuerdo\s+con\s+[^,]+,',
            r'los\s+datos\s+(?:muestran|indican|demuestran)\s+[^.]+',
            r'la\s+evidencia\s+(?:muestra|indica|demuestra)\s+[^.]+'
        ]

        for pattern in evidence_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            rhetorical['evidence'].extend(matches[:50])

        # Patrones de ejemplos
        example_patterns = [
            r'por\s+ejemplo[^.]+',
            r'como\s+es\s+el\s+caso\s+de\s+[^.]+',
            r'tal\s+como\s+[^.]+'
        ]

        for pattern in example_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            rhetorical['examples'].extend(matches[:50])
```

```
11623:
11624:         # Patrones de conclusiones
11625:         conclusion_patterns = [
11626:             r'en\s+conclusiÃ³n[^.]+',
11627:             r'por\s+lo\s+tanto[^.]+',
11628:             r'en\s+sÃ-ntesis[^.]+'
11629:         ]
11630:
11631:         for pattern in conclusion_patterns:
11632:             matches = re.findall(pattern, text, re.IGNORECASE)
11633:             rhetorical['conclusions'].extend(matches[:50])
11634:
11635:         return rhetorical
11636:
11637:     def _analyze_information_flow(self, text: str) -> Dict[str, Any]:
11638:         """
11639:         Analyze information flow patterns in text.
11640:
11641:         Inputs:
11642:             text (str): Text to analyze
11643:         Outputs:
11644:             Dict[str, Any]: Flow analysis metrics
11645:         """
11646:         sentences = filter_empty_sentences(re.split(r'[.!?]+\s+', text))
11647:         sentences = [s for s in sentences if len(s) > 20]
11648:         if not sentences:
11649:             return {'sentence_count': 0}
11650:
11651:         sentence_lengths = [len(s.split()) for s in sentences]
11652:         words = text.lower().split()
11653:         unique_words = set(words)
11654:
11655:         return {
11656:             'sentence_count': len(sentences),
11657:             'avg_sentence_length': np.mean(sentence_lengths),
11658:             'std_sentence_length': np.std(sentence_lengths),
11659:             'lexical_diversity': len(unique_words) / len(words) if words else 0,
11660:             'total_words': len(words),
11661:             'unique_words': len(unique_words)
11662:         } #
11663:
11664:     def _extract_document_structure(self, text: str) -> Dict[str, Any]:
11665:         """AnÃ¡lisis estructural del documento"""
11666:         return {
11667:             'section_hierarchy': self._identify_section_hierarchy(text),
11668:             'policy_frameworks': self._identify_policy_frameworks(text),
11669:             'raw_text': text # Almacenar texto para referencia
11670:         }
11671:
11672:     def _identify_section_hierarchy(self, text: str) -> List[Dict[str, Any]]:
11673:         """Identificar la jerarquÃ-a de secciones/tÃ-tulos"""
11674:         hierarchy = []
11675:         # Patrones de encabezados (simplificados)
11676:         patterns = [
11677:             (r'^(CAPÃ\215TULO\s+I[VX]+\s*:\s*.+)$', 1, 'chapter'),
11678:             (r'^(SECCIÃ\223N\s+[A-Z]\s*:\s*.+)$', 2, 'section'),
```

```
            (r'^\d+\.\d+(?:\.\d+)?\s+([A-ZÃ\221Ã\201Ã\211Ã\215Ã\223Ã\232][^.!?]+)$', 3, 'header_numbered'),
            (r'^([a-z]\))\s+([A-ZÃ\221Ã\201Ã\211Ã\215Ã\223Ã\232][^.!?]+)$', 4, 'item_alpha'),
            (r'^â\200¢\s+([A-ZÃ\221Ã\201Ã\211Ã\215Ã\223Ã\232][^.!?]+)$', 5, 'bullet'),
        ]

        lines = text.split('\n')
        for idx, line in enumerate(lines):
            line = line.strip()
            if not line or len(line) < 3: continue

            for pattern, level, header_type in patterns:
                match = re.match(pattern, line, re.MULTILINE)
                if match:
                    title = match.group(2) if match.lastindex >= 2 else match.group(1)
                    hierarchy.append({
                        'title': title.strip(),
                        'level': level,
                        'line_number': idx,
                        'type': header_type,
                        'full_text': line
                    })
                    break
        return hierarchy

    def _identify_policy_frameworks(self, text: str) -> List[Dict[str, Any]]:
        """Identificar todos los marcos de polÃ-tica presentes"""
        frameworks = []
        framework_patterns = [
            (r'plan\s+(?:de\s+)?desarrollo\s+(?:municipal|departamental|nacional)?', 'plan_desarrollo'),
            (r'polÃ-tica\s+pÃºblica\s+(?:de\s+)?[\w\s]+', 'politica_publica'),
            (r'ley\s+[\w\s]+', 'legal_framework')
        ]

        for pattern, framework_type in framework_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                frameworks.append({
                    'text': match.group(0),
                    'type': framework_type,
                    'position': match.span()
                })
        return frameworks

    def _extract_content_for_pa_dimension(
        self,
        document_text: str,
        policy_area: str,
        dimension: str,
        sentences: List[str],
        sentence_positions: List[Tuple[int, int]],
        sentence_embeddings: Optional[np.ndarray] = None
    ) -> Dict[str, Any]:
        """
        Extract most relevant content for a specific (PA, DIM) combination.

        Uses embedding-based similarity to find sentences most aligned with
```

```
        the policy area and dimension keywords.

        Args:
            document_text: Full document text
            policy_area: Policy area code (PA01-PA10)
            dimension: Dimension code (DIM01-DIM06)
            sentences: List of document sentences
            sentence_positions: List of (start, end) byte positions for each sentence
            sentence_embeddings: Pre-computed embeddings (optional, computed if None)

        Returns:
            Dictionary with segment metadata and text
        """
        # Generate query embedding from PA + DIM keywords
        pa_keywords = self._pa_keywords.get(policy_area, [])
        dim_keywords = self._dim_keywords.get(dimension, [])
        query_text = " ".join(pa_keywords + dim_keywords)

        # Get query embedding
        query_embedding = self._spc_sem.embed_text(query_text)

        # Compute sentence embeddings if not provided
        if sentence_embeddings is None:
            sentence_embeddings = self._spc_sem.embed_batch(sentences)

        # Compute cosine similarity between query and all sentences
        from sklearn.metrics.pairwise import cosine_similarity
        similarities = cosine_similarity(
            query_embedding.reshape(1, -1),
            sentence_embeddings
        )[0]

        # Find top-K most similar sentences (K=10)
        top_k = min(10, len(sentences))
        top_indices = np.argsort(similarities)[-top_k:][::-1]

        # Extract contiguous region around top sentences
        if len(top_indices) == 0:
            # Fallback: return first 800 chars
            segment_text = document_text[:800]
            segment_start = 0
            segment_end = len(segment_text)
        else:
            # Find min/max positions to create contiguous chunk
            min_idx = min(top_indices)
            max_idx = max(top_indices)

            # Expand window by Â±2 sentences for context
            start_idx = max(0, min_idx - 2)
            end_idx = min(len(sentences) - 1, max_idx + 2)

            # Get byte positions
            segment_start = sentence_positions[start_idx][0]
            segment_end = sentence_positions[end_idx][1]
            segment_text = document_text[segment_start:segment_end]

```

```
        # Compute relevance score (mean of top-K similarities)
        relevance_score = float(np.mean(similarities[top_indices])) if len(top_indices) > 0 else 0.0

        return {
            "text": segment_text,
            "position": (segment_start, segment_end),
            "policy_area": policy_area,
            "dimension": dimension,
            "relevance_score": relevance_score,
            "top_sentence_indices": top_indices.tolist(),
            "query_keywords": pa_keywords + dim_keywords
        }

    def _generate_60_structured_segments(
        self,
        document_text: str,
        structural_analysis: Dict[str, Any]
    ) -> List[Dict[str, Any]]:
        """
        Generate EXACTLY 60 structured segments aligned by (PA Ã\227 DIM) matrix.

        This replaces FASE 4 semantic segmentation with structured extraction.
        Each segment is explicitly aligned to one Policy Area and one Dimension.

        Args:
            document_text: Full document text
            structural_analysis: Output from FASE 3 (document structure analysis)

        Returns:
            List of 60 segment dictionaries, one per (PA, DIM) combination
        """
        from farfan_pipeline.core.canonical_notation import get_all_policy_areas, get_all_dimensions

        self.logger.info("FASE 4: Generating 60 structured segments (PA Ã\227 DIM matrix)")

        # Split document into sentences for embedding-based extraction
        sentences = sent_tokenize(document_text, language='spanish')

        # Compute sentence positions
        sentence_positions = []
        current_pos = 0
        for sent in sentences:
            start = document_text.find(sent, current_pos)
            end = start + len(sent)
            sentence_positions.append((start, end))
            current_pos = end

        # Pre-compute sentence embeddings (once for all 60 extractions)
        self.logger.info(f"Computing embeddings for {len(sentences)} sentences...")
        sentence_embeddings = self._spc_sem.embed_batch(sentences)

        # Generate 60 segments
        policy_areas = get_all_policy_areas()  # PA01..PA10
        dimensions = get_all_dimensions()      # D1..D6

        structured_segments = []
```

```
11847:
11848:         for pa_code, pa_info in policy_areas.items():
11849:             for dim_key, dim_info in dimensions.items():
11850:                 segment = self._extract_content_for_pa_dimension(
11851:                     document_text=document_text,
11852:                     policy_area=pa_code,
11853:                     dimension=dim_info.code,
11854:                     sentences=sentences,
11855:                     sentence_positions=sentence_positions,
11856:                     sentence_embeddings=sentence_embeddings
11857:                 )
11858:
11859:                 # Add PA and DIM metadata
11860:                 segment['policy_area_id'] = pa_code
11861:                 segment['dimension_id'] = dim_info.code
11862:                 segment['pa_name'] = pa_info.name
11863:                 segment['dim_label'] = dim_info.label
11864:
11865:                 structured_segments.append(segment)
11866:
11867:                 self.logger.debug(
11868:                     f"  Extracted segment for {pa_code} Ã\227 {dim_info.code}: "
11869:                     f"{len(segment['text'])} chars, relevance={segment['relevance_score']:.3f}"
11870:                 )
11871:
11872:         assert len(structured_segments) == 60, \
11873:             f"Expected 60 segments, got {len(structured_segments)}"
11874:
11875:         self.logger.info(f"â\234\205 Generated exactly {len(structured_segments)} structured segments")
11876:
11877:         return structured_segments
11878:
11879:     def generate_smart_chunks(self, document_text: str, document_metadata: Dict) -> List[SmartPolicyChunk]:
11880:         """
11881:         Main pipeline phase: Generate 60 Smart Policy Chunks aligned with the P01-ES v1.0 spec.
11882:
11883:         This streamlined pipeline executes the structured (Policy Area x Dimension)
11884:         segmentation and bypasses the subsequent complex analysis to ensure the
11885:         output is exactly the 60 thematically isolated chunks required by the orchestrator.
11886:         """
11887:         self.logger.info("Starting Smart Chunks v3.0 pipeline (P01-ES v1.0 Compliant Mode)")
11888:
11889:         # FASE 1: Preprocesamiento avanzado
11890:         normalized_text = self._advanced_preprocessing(document_text)
11891:
11892:         # FASE 2: AnÃ¡lisis estructural y de jerarquÃ-a (needed for segmentation context)
11893:         structural_analysis = self._extract_document_structure(normalized_text)
11894:         structural_analysis['raw_text'] = document_text
11895:
11896:         # FASE 4: Structured (PA Ã\227 DIM) segmentation - EXACTLY 60 chunks
11897:         structured_segments = self._generate_60_structured_segments(
11898:             document_text=document_text,
11899:             structural_analysis=structural_analysis
11900:         )
11901:         self.logger.info(f"â\234\205 Generated {len(structured_segments)} structured segments (PA Ã\227 DIM)")
11902:
```

```
11903:         # Convert the 60 segments to SmartPolicyChunk objects
11904:         smart_chunks = []
11905:         document_id = document_metadata.get("document_id", "doc_001")
11906:
11907:         for segment in structured_segments:
11908:             text = segment.get("text", "")
11909:             content_hash = hashlib.sha256(text.encode('utf-8')).hexdigest()
11910:             pa_id = segment.get("policy_area_id")
11911:             dim_id = segment.get("dimension_id")
11912:             chunk_id = f"{document_id}_{pa_id}_{dim_id}_{content_hash[:8]}"
11913:
11914:             chunk = SmartPolicyChunk(
11915:                 chunk_id=chunk_id,
11916:                 document_id=document_id,
11917:                 content_hash=content_hash,
11918:                 text=text,
11919:                 normalized_text=self._advanced_preprocessing(text),
11920:                 policy_area_id=pa_id,
11921:                 dimension_id=dim_id,
11922:                 document_position=segment.get("position", (0, 0)),
11923:                 # --- Default empty values for bypassed analysis ---
11924:                 semantic_density=0.0,
11925:                 section_hierarchy=[],
11926:                 chunk_type=ChunkType.MIXTO,
11927:                 causal_chain=[],
11928:                 policy_entities=[],
11929:                 implicit_assumptions=[],
11930:                 contextual_presuppositions=[],
11931:                 confidence_metrics={'overall_confidence': segment.get('relevance_score', 0.0)},
11932:                 coherence_score=segment.get('relevance_score', 0.0),
11933:                 strategic_importance=segment.get('relevance_score', 0.0),
11934:             )
11935:             smart_chunks.append(chunk)
11936:
11937:         self.logger.info(f"Pipeline completed: {len(smart_chunks)} chunks generated.")
11938:         return smart_chunks
11939:
11940:     def _advanced_preprocessing(self, text: str) -> str:
11941:         """
11942:         Advanced text preprocessing with normalization and encoding fixes.
11943:
11944:         Inputs:
11945:             text (str): Raw input text
11946:         Outputs:
11947:             str: Normalized and cleaned text
11948:         """
11949:         # NormalizaciÃ³n de espacios y saltos de lÃnea
11950:         text = re.sub(r'\s+', ' ', text)
11951:         text = re.sub(r'\n+', '\n', text)
11952:
11953:         # CorrecciÃ³n de encodings problemÃ¡ticos (common UTF-8 mojibake)
11954:         encoding_fixes = {
11955:             'Ã\203Â¡': 'Ã¡', 'Ã\203Â©': 'Ã©', 'Ã\203Â': 'Ã', 'Ã\203Â³': 'Ã³', 'Ã\203Âº': 'Ãº',
11956:             'Ã\203Â±': 'Ã±', 'Ã\203': 'Ã\221', 'Ã\203 ': 'Ã ', 'Ã\203Â¨': 'Ã¨', 'Ã\203Â¬': 'Ã¬',
11957:             'Ã\203Â²': 'Ã²', 'Ã\203Â¹': 'Ã¹', 'Ã\203Â¼': 'Ã¼', 'Ã\203': 'Ã\201', 'Ã\203â\200°': 'Ã\211',
11958:             'Ã\203': 'Ã\215', 'Ã\203"': 'Ã\223', 'Ã\203Å¡': 'Ã\232'
```

```
        }
        for wrong, correct in encoding_fixes.items():
            text = text.replace(wrong, correct)

        return text

    def _enrich_with_inter_chunk_relationships(self, chunks: List[SmartPolicyChunk]) -> List[SmartPolicyChunk]:
        """
        CANONICAL SOTA: Enrich chunks with semantic relationships using BGE-M3 embeddings.

        Uses batch embeddings and vectorized similarity computation instead of
        manual loops and sklearn.cosine_similarity.
        """
        if not chunks:
            return []

        texts = [c.text for c in chunks]
        self.corpus_embeddings = self._generate_embeddings_for_corpus(texts)

        # Vectorized cosine similarity (no sklearn dependency)
        norms = np.linalg.norm(self.corpus_embeddings, axis=1, keepdims=True)
        normalized_embs = self.corpus_embeddings / (norms + 1e-8)
        similarity_matrix = np.dot(normalized_embs, normalized_embs.T)

        # Efficiently find related chunks using vectorized operations
        for i, chunk in enumerate(chunks):
            # Get similarities for this chunk (excluding self)
            sims = similarity_matrix[i].copy()
            sims[i] = -1  # Exclude self

            # Find chunks above threshold
            related_indices = np.where(sims >= self.config.CROSS_REFERENCE_MIN_SIMILARITY)[0]

            # Sort by similarity
            sorted_indices = related_indices[np.argsort(-sims[related_indices])]

            chunk.related_chunks = [
                (chunks[j].chunk_id, float(sims[j]))
                for j in sorted_indices
            ]

        return chunks

    def _generate_embeddings_for_corpus(self, texts: List[str], batch_size: int = 64) -> np.ndarray:
        """
        CANONICAL SOTA: Batch embeddings using SemanticChunkingProducer.

        SemanticChunkingProducer handles batching internally with BGE-M3.
        batch_size kept for signature compatibility.

        Inputs:
            texts (List[str]): List of text strings to embed
            batch_size (int): Batch size hint (default: 64)
        Outputs:
            np.ndarray: Array of embeddings, shape (n_texts, embedding_dim)
        """
```

```
        if not texts:
            return np.array([])

        # CANONICAL SOTA: Use SemanticChunkingProducer batch embedding
        embs = self._spc_sem.embed_batch(texts)
        return np.vstack(embs).astype(np.float32)

    def _validate_strategic_integrity(self, chunks: List[SmartPolicyChunk]) -> List[SmartPolicyChunk]:
        """Validar que los chunks cumplan con umbrales mÃ-nimos de calidad y completitud"""
        validated = []
        for chunk in chunks:
            # Criterios de validaciÃ³n
            is_valid = (
                chunk.coherence_score >= self.config.MIN_COHERENCE_SCORE and
                chunk.completeness_index >= self.config.MIN_COMPLETENESS_INDEX and
                chunk.strategic_importance >= self.config.MIN_STRATEGIC_IMPORTANCE and
                chunk.information_density >= self.config.MIN_INFORMATION_DENSITY and
                len(chunk.text) >= self.config.MIN_CHUNK_SIZE
            )

            if is_valid:
                validated.append(chunk)
            else:
                self.logger.warning(f"Chunk {chunk.chunk_id} no pasÃ³ validaciÃ³n de integridad")

        return validated

    def _intelligent_deduplication(self, chunks: List[SmartPolicyChunk]) -> List[SmartPolicyChunk]:
        """DeduplicaciÃ³n inteligente de chunks"""
        if len(chunks) < 2:
            return chunks

        deduplicated = []
        processed_hashes = set()

        for chunk in chunks:
            # Verificar por hash exacto
            if chunk.content_hash not in processed_hashes:
                processed_hashes.add(chunk.content_hash)
                deduplicated.append(chunk)

        # DeduplicaciÃ³n semÃ¡ntica (para near-duplicates) using vectorized ops
        final_list = []
        if deduplicated:
            embeddings = self._generate_embeddings_for_corpus([c.text for c in deduplicated])
            n_chunks = len(deduplicated)

            # Vectorized cosine similarity (no sklearn dependency)
            norms = np.linalg.norm(embeddings, axis=1, keepdims=True)
            normalized_embs = embeddings / (norms + 1e-8)
            sim_matrix = np.dot(normalized_embs, normalized_embs.T)

            is_duplicate = [False] * n_chunks

            for i in range(n_chunks):
                if is_duplicate[i]:
```

```
                    continue

                final_list.append(deduplicated[i])

                for j in range(i + 1, n_chunks):
                    if sim_matrix[i, j] >= self.config.DEDUPLICATION_THRESHOLD:
                        is_duplicate[j] = True

        return final_list

    def _rank_by_strategic_importance(self, chunks: List[SmartPolicyChunk]) -> List[SmartPolicyChunk]:
        """Ranking final de chunks por importancia estratÃ©gica"""
        # Usar el campo 'strategic_importance' calculado en _create_smart_policy_chunk
        chunks.sort(key=lambda x: x.strategic_importance, reverse=True)
        return chunks

    # --- MÃ©todos de EvaluaciÃ³n (ContinuaciÃ³n de smart_policy_chunks_industrial_v3_part4_completion.py) ---

    def _assess_strategic_importance(
        self,
        strategic_unit: Dict,
        causal_evidence: List[CausalEvidence],
        policy_entities: List[PolicyEntity]
    ) -> float:
        """Evaluar importancia estratÃ©gica"""
        factors = {
            'causal_strength': np.mean([e.strength_score for e in causal_evidence]) if causal_evidence else 0,
            'entity_relevance': len([e for e in policy_entities if e.context_role in [PolicyEntityRole.EXECUTOR, PolicyEntityRole.BENEFICIARY]]) / max(len(p
olicy_entities), 1),
            'position_weight': 1.0 if strategic_unit.get('position', (0, 0))[0] < 1000 else 0.5, # Ponderar texto temprano
            'keyword_presence': self._calculate_strategic_keyword_presence(strategic_unit.get('text', ''))
        }

        return np.mean(list(factors.values()))

    def _calculate_strategic_keyword_presence(self, text: str) -> float:
        """Calcular presencia de palabras clave estratÃ©gicas"""
        strategic_keywords = [
            'objetivo', 'meta', 'estrategia', 'prioridad', 'desarrollo',
            'transformaciÃ³n', 'impacto', 'resultado', 'sostenible', 'integral'
        ]

        text_lower = text.lower()
        keyword_count = sum(1 for kw in strategic_keywords if kw in text_lower)

        return min(keyword_count / len(strategic_keywords), 1.0)

    def _calculate_information_density(self, text: str) -> float:
        """Calcular densidad de informaciÃ³n"""
        if not text:
            return 0.0

        # MÃ©tricas de densidad
        words = text.split()
        sentences = re.split(r'[.!?]+', text)

```

```
12126:         if not words or not sentences:
12127:             return 0.0
12128:
12129:         metrics = {
12130:             'lexical_diversity': len(set(words)) / len(words),
12131:             'avg_sentence_length': np.mean([len(s.split()) for s in sentences if s.strip()]),
12132:             'entity_density': len(re.findall(r'\b[A-Z][a-z]+\b', text)) / len(words) # EstimaciÃ³n de entidades
12133:         }
12134:
12135:         return np.mean(list(metrics.values()))
12136:
12137:     def _assess_actionability(self, text: str, policy_entities: List[PolicyEntity]) -> float:
12138:         """Evaluar accionabilidad del chunk"""
12139:         action_indicators = [
12140:             r'se\s+(?:debe|deberÃ¡|requiere)',
12141:             r'es\s+necesario',
12142:             r'(?:implementar|ejecutar|desarrollar|crear|establecer|asignar)\s+',
12143:             r'(?:el|la)\s+(?:Ministerio|AlcaldÃ-a|DirecciÃ³n)\s+(?:debe|deberÃ¡)'
12144:         ]
12145:
12146:         action_score = sum(len(re.findall(pat, text, re.IGNORECASE)) for pat in action_indicators)
12147:
12148:         # Presencia de ejecutores explÃ-citos
12149:         executor_count = len([e for e in policy_entities if e.context_role == PolicyEntityRole.EXECUTOR])
12150:
12151:         final_score = (min(action_score / 5.0, 1.0) * 0.7) + (min(executor_count / 2.0, 1.0) * 0.3)
12152:         return final_score
12153:
12154: # ============================================================================
12155: # CANONICAL PIPELINE WRAPPER
12156: # ============================================================================
12157:
12158: class CPPIngestionPipeline:
12159:     """
12160:     Canonical wrapper for the SPC ingestion process.
12161:     Provides the interface expected by run_policy_pipeline_verified.py.
12162:     """
12163:     def __init__(self):
12164:         self.system = StrategicChunkingSystem()
12165:         self.logger = logging.getLogger("CPPIngestionPipeline")
12166:
12167:     async def process(self, pdf_path: Path) -> Any:
12168:         """
12169:         Process a PDF document into a CanonPolicyPackage (SPC output).
12170:
12171:         Args:
12172:             pdf_path: Path to the input PDF file.
12173:
12174:         Returns:
12175:             The processed CanonPolicyPackage (or equivalent SPC object).
12176:         """
12177:         self.logger.info(f"Starting CPPIngestionPipeline for: {pdf_path}")
12178:
12179:         if not pdf_path.exists():
12180:             raise FileNotFoundError(f"Input PDF not found: {pdf_path}")
12181:
```

```
        # Extract text with provenance (page numbers)
        # Using pdfplumber as per SOTA requirements
        document_text = ""
        page_map = [] # List of (start_char, end_char, page_number)

        try:
            import pdfplumber
            with pdfplumber.open(pdf_path) as pdf:
                current_char = 0
                for i, page in enumerate(pdf.pages):
                    text = page.extract_text() or ""
                    # Normalize text slightly to avoid issues
                    text = self.system._advanced_preprocessing(text)

                    start = current_char
                    end = start + len(text)
                    page_number = i + 1

                    page_map.append({
                        "start": start,
                        "end": end,
                        "page": page_number,
                        "bbox": page.bbox
                    })

                    document_text += text + "\n" # Add newline as separator
                    current_char = len(document_text)

        except ImportError:
            self.logger.warning("pdfplumber not installed. Fallback to basic text reading (NO PROVENANCE).")
            # Fallback for dev/testing without pdfplumber
            try:
                document_text = pdf_path.read_text(encoding='utf-8')
            except UnicodeDecodeError:
                 # Try latin-1 as fallback
                document_text = pdf_path.read_text(encoding='latin-1')

        # Generate chunks using the core system
        # Note: We pass the full text. The system currently doesn't natively handle
        # the page_map during chunk generation, so we need to inject provenance *after*
        # chunking but *before* returning.

        # Create a mock metadata dict
        metadata = {
            "source_file": str(pdf_path),
            "ingestion_timestamp": canonical_timestamp()
        }

        # Run the core pipeline
        chunks = self.system.generate_smart_chunks(document_text, metadata)

        # POST-PROCESSING: Inject Provenance
        # Map chunks back to pages using the page_map
        if page_map:
            self._inject_provenance(chunks, page_map, str(pdf_path))
        else:
```

```
12238:              self.logger.warning("No page map available. Provenance will be empty.")
12239:
12240:         # Construct the return object (mimicking CanonPolicyPackage structure expected by adapter)
12241:         # The adapter expects an object with a .chunk_graph.chunks attribute
12242:
12243:         @dataclass
12244:         class CanonPolicyPackageShim:
12245:             chunk_graph: Any
12246:             schema_version: str = "SPC-2025.1"
12247:             metadata: Dict = field(default_factory=dict)
12248:             quality_metrics: Any = None
12249:             policy_manifest: Any = None
12250:
12251:         @dataclass
12252:         class ChunkGraphShim:
12253:             chunks: Dict[str, Any]
12254:             edges: List = field(default_factory=list)
12255:
12256:         # Convert list of chunks to dict keyed by ID
12257:         chunks_dict = {c.chunk_id: c for c in chunks}
12258:
12259:         return CanonPolicyPackageShim(
12260:             chunk_graph=ChunkGraphShim(chunks=chunks_dict),
12261:             metadata={"spc_rich_data": True}
12262:         )
12263:
12264:     def _inject_provenance(self, chunks: List[SmartPolicyChunk], page_map: List[Dict], source_file: str):
12265:         """
12266:         Map chunks to pages based on their text content or position.
12267:         Since generate_smart_chunks might lose exact character offsets relative to the *original*
12268:         concatenated string (due to preprocessing), we might need a robust matching strategy.
12269:
12270:         However, StrategicChunkingSystem usually preserves text.
12271:         We will assume 'document_position' in SmartPolicyChunk refers to the character offsets
12272:         in the passed 'document_text'.
12273:         """
12274:         for chunk in chunks:
12275:             if not chunk.document_position:
12276:                 continue
12277:
12278:             start_char, end_char = chunk.document_position
12279:
12280:             # Find which page(s) this span covers
12281:             # We assign the page where the chunk *starts* as the primary page
12282:             primary_page = None
12283:             for page_info in page_map:
12284:                 # If chunk starts within this page
12285:                 if page_info["start"] <= start_char < page_info["end"]:
12286:                     primary_page = page_info["page"]
12287:                     break
12288:
12289:             # If not found (e.g. due to newline insertion), try to find where it overlaps most
12290:             if not primary_page:
12291:                  for page_info in page_map:
12292:                     if (start_char < page_info["end"] and end_char > page_info["start"]):
12293:                          primary_page = page_info["page"]
```

```
                        break

            if primary_page:
                chunk.provenance = Provenance(
                    page_number=primary_page,
                    section_header=None, # Could be extracted if we had section info
                    source_file=source_file
                )

# ==============================================================================
# SCRIPT DE EJECUCIÃ\223N (MAIN)
# ==============================================================================

def validate_cli_arguments(args):
    """
    Validate CLI arguments before processing.

    Returns:
        Tuple[Path, Path]: Normalized input and output paths
    """
    input_path = Path(args.input).expanduser()
    if not input_path.exists():
        raise ValidationError(f"Input file not found: {input_path}")

    output_path = Path(args.output).expanduser()
    output_dir = output_path.parent if output_path.name else output_path
    if not output_dir.exists():
        raise ValidationError(f"Output directory does not exist: {output_dir}")
    if not os.access(output_dir, os.W_OK):
        raise ValidationError(f"Output directory is not writable: {output_dir}")

    if not args.doc_id or not args.doc_id.strip():
        raise ValidationError("Document ID cannot be empty")

    if args.max_chunks < 0:
        raise ValidationError(f"max_chunks must be non-negative, got: {args.max_chunks}")

    logger.info("CLI arguments validated successfully")
    return input_path, output_path

def main(args):
    """
    Main pipeline function for Smart Policy Chunks generation.

    Inputs:
        args: Command-line arguments with input/output paths and configuration
    Outputs:
        int: Exit code (0 for success, 1 for error)
    """
    try:
        # 0. Validate CLI arguments
        input_path, output_path = validate_cli_arguments(args)

        # 1. Cargar el documento
        logger.info(f"Loading input document: {input_path}")
        try:
```

```
            document_text = input_path.read_text(encoding='utf-8')
        except IOError as e:
            raise ProcessingError(f"Failed to read input file: {e}")

        # 2. Inicializar el sistema (con lazy loading)
        logger.info("Initializing Strategic Chunking System...")
        chunking_system = StrategicChunkingSystem()

        # Metadata del documento
        metadata = {
            'document_id': args.doc_id,
            'title': args.title,
            'version': 'v3.0',
            'processing_timestamp': canonical_timestamp()
        }

        # 3. Generar Smart Chunks
        logger.info("Generating smart policy chunks...")
        chunks = chunking_system.generate_smart_chunks(document_text, metadata)

        # 4. Limitar el nÃºmero de chunks a guardar si se especifica
        if args.max_chunks > 0:
            logger.info(f"Limiting output to {args.max_chunks} chunks")
            chunks = chunks[:args.max_chunks]

        # 5. Serializar resultados (summary version with truncated text)
        logger.info(f"Generated {len(chunks)} chunks. Saving results...")
        output_data = {
            'metadata': metadata,
            'config': {
                'min_chunk_size': chunking_system.config.MIN_CHUNK_SIZE,
                'max_chunk_size': chunking_system.config.MAX_CHUNK_SIZE,
                'semantic_coherence_threshold': chunking_system.config.SEMANTIC_COHERENCE_THRESHOLD
            },
            'chunks': [
                asdict(c) for c in chunks
            ]
        }

        # Truncar el texto del chunk de forma segura para summary.json
        for chunk_data in output_data['chunks']:
            original_text = chunk_data['text']
            # Safe UTF-8 truncation to 200 bytes
            chunk_data['text'] = safe_utf8_truncate(original_text, 200)
            if len(original_text.encode('utf-8')) > 200:
                chunk_data['text'] += '...'

        summary_payload = json.dumps(output_data, indent=2, ensure_ascii=False, default=np_to_list)
        try:
            output_path.write_text(summary_payload, encoding='utf-8')
            logger.info(f"Summary output saved to: {output_path}")
        except (IOError, TypeError) as e:
            raise SerializationError(f"Failed to write summary output: {e}")

        # 6. Guardar versiÃ³n completa sin truncar texto (full.json)
        if args.save_full:
```

```
            full_output = output_path.with_name(f"{output_path.stem}_full.json")
            try:
                # Crear una estructura completa con texto Ã-ntegro
                full_data = {
                    'metadata': metadata,
                    'config': output_data['config'],
                    'chunks': []
                }

                for chunk in chunks:
                    chunk_dict = asdict(chunk)
                    # Mantener texto completo
                    full_data['chunks'].append(chunk_dict)

                full_payload = json.dumps(full_data, indent=2, ensure_ascii=False, default=np_to_list)
                full_output.write_text(full_payload, encoding='utf-8')
                logger.info(f"Full output saved to: {full_output}")
            except (IOError, TypeError) as e:
                raise SerializationError(f"Failed to write full output: {e}")

        # 7. Generar reporte de verificaciÃ³n con canonical timestamps
        verification = {
            'pipeline_version': 'SMART-CHUNK-3.0-FINAL',
            'execution_timestamp': canonical_timestamp(),
            'input_file': args.input,
            'input_hash': hashlib.sha256(document_text.encode('utf-8')).hexdigest(),
            'output_hash': hashlib.sha256(summary_payload.encode('utf-8')).hexdigest(),
            'chunks_generated': len(chunks),
            'validation_passed': all(
                c.coherence_score >= chunking_system.config.MIN_COHERENCE_SCORE
                for c in chunks
            ),
            'success': len(chunks) > 0 and all(c.coherence_score >= chunking_system.config.MIN_COHERENCE_SCORE for c in chunks)
        }

        verification_file = output_path.with_name(f"{output_path.stem}_verification.json")
        verification_payload = json.dumps(verification, indent=2, default=np_to_list)
        try:
            verification_file.write_text(verification_payload, encoding='utf-8')
            logger.info(f"Verification report saved to: {verification_file}")
        except IOError as e:
            logger.warning(f"Failed to write verification file: {e}")

        if verification['success']:
            logger.info("Pipeline completed successfully")
            return 0
        else:
            logger.warning("Pipeline completed with validation warnings")
            return 0

    except ValidationError as e:
        logger.error(f"Validation error: {e}")
        return 1
    except ProcessingError as e:
        logger.error(f"Processing error: {e}")
        return 1
```

```
12462:     except SerializationError as e:
12463:         logger.error(f"Serialization error: {e}")
12464:         return 1
12465:     except Exception as e:
12466:         logger.error(f"Unexpected error in pipeline: {e}")
12467:         import traceback
12468:         traceback.print_exc()
12469:         return 1
12470:
12471: if __name__ == '__main__':
12472:     import argparse
12473:     import sys
12474:
12475:     # ConfiguraciÃ³n de argumentos CLI
12476:     parser = argparse.ArgumentParser(
12477:         description="Smart Policy Chunks Pipeline v3.0 - Industrial Grade",
12478:         formatter_class=argparse.RawDescriptionHelpFormatter,
12479:         epilog="""
12480: Examples:
12481:   python smart_policy_chunks_canonic_phase_one.py --input plan.pdf --output chunks.json
12482:   python smart_policy_chunks_canonic_phase_one.py --input plan.pdf --output chunks.json --save_full
12483:         """
12484:     )
12485:     parser.add_argument(
12486:         '--input',
12487:         type=str,
12488:         required=True,
12489:         help='Path to input policy document (required)'
12490:     )
12491:     parser.add_argument(
12492:         '--output',
12493:         type=str,
12494:         required=False,
12495:         default='output_chunks.json',
12496:         help='Path to output JSON file (default: output_chunks.json)'
12497:     )
12498:     parser.add_argument(
12499:         '--doc_id',
12500:         type=str,
12501:         required=False,
12502:         default='POL_PLAN_001',
12503:         help='Document identifier (default: POL_PLAN_001)'
12504:     )
12505:     parser.add_argument(
12506:         '--title',
12507:         type=str,
12508:         required=False,
12509:         default='Plan de Desarrollo',
12510:         help='Document title (default: Plan de Desarrollo)'
12511:     )
12512:     parser.add_argument(
12513:         '--max_chunks',
12514:         type=int,
12515:         required=False,
12516:         default=0,
12517:         help='Maximum number of chunks to generate (0 = unlimited, default: 0)'
```

```
    )
    parser.add_argument(
        '--save_full',
        action='store_true',
        help='Save full version with complete text (creates *_full.json)'
    )

    # Parse arguments - allow argparse to exit normally on error
    args = parser.parse_args()

    # Execute main pipeline
    exit_code = main(args)
    sys.exit(exit_code)


================================================================================
FILE: src/farfan_pipeline/processing/structural.py
================================================================================

"""
Structural normalization with policy-awareness.

Segments documents into policy-aware units.
"""

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


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 ""


================================================================================
FILE: src/farfan_pipeline/processing/uncertainty_quantification.py
================================================================================

"""
Bayesian Uncertainty Quantification for Aggregation

This module implements rigorous uncertainty propagation for the aggregation pipeline.
It provides confidence intervals, epistemic/aleatoric decomposition, and sensitivity analysis.

Methods:
- Bootstrap resampling for empirical confidence intervals
- Bayesian error propagation for weighted averages
- Variance decomposition (epistemic vs aleatoric uncertainty)
- Monte Carlo sampling for non-linear aggregations

References:
- Efron, B. (1979). "Bootstrap methods: another look at the jackknife"
- Saltelli, A. et al. (2020). "Five ways to ensure that models serve society"
- Kendall, A. & Gal, Y. (2017). "What uncertainties do we need in Bayesian deep learning?"
"""

from __future__ import annotations

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

import numpy as np
from scipy import stats

logger = logging.getLogger(__name__)


@dataclass(frozen=True)
class UncertaintyMetrics:
    """
    Comprehensive uncertainty metrics for a score.

    Attributes:
        mean: Point estimate (expected value)
        std: Standard deviation
        variance: Variance
        confidence_interval_95: 95% CI tuple (lower, upper)
        confidence_interval_99: 99% CI tuple (lower, upper)
        epistemic_uncertainty: Model uncertainty (reducible with more data)
        aleatoric_uncertainty: Inherent data noise (irreducible)
        coefficient_of_variation: std/mean (relative uncertainty)
        skewness: Distribution asymmetry
        kurtosis: Distribution tail heaviness
```

```
        metadata: Additional diagnostic information
    """
    mean: float
    std: float
    variance: float
    confidence_interval_95: tuple[float, float]
    confidence_interval_99: tuple[float, float]
    epistemic_uncertainty: float
    aleatoric_uncertainty: float
    coefficient_of_variation: float
    skewness: float
    kurtosis: float
    metadata: dict[str, Any] = field(default_factory=dict)

    def is_high_uncertainty(self, threshold: float = 0.3) -> bool:
        """Check if coefficient of variation exceeds threshold."""
        return self.coefficient_of_variation > threshold

    def dominant_uncertainty_type(self) -> str:
        """Identify whether epistemic or aleatoric dominates."""
        if self.epistemic_uncertainty > self.aleatoric_uncertainty * 1.2:
            return "epistemic"
        elif self.aleatoric_uncertainty > self.epistemic_uncertainty * 1.2:
            return "aleatoric"
        else:
            return "balanced"

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        return {
            "mean": self.mean,
            "std": self.std,
            "variance": self.variance,
            "confidence_interval_95": self.confidence_interval_95,
            "confidence_interval_99": self.confidence_interval_99,
            "epistemic_uncertainty": self.epistemic_uncertainty,
            "aleatoric_uncertainty": self.aleatoric_uncertainty,
            "coefficient_of_variation": self.coefficient_of_variation,
            "skewness": self.skewness,
            "kurtosis": self.kurtosis,
            "is_high_uncertainty": self.is_high_uncertainty(),
            "dominant_uncertainty": self.dominant_uncertainty_type(),
            "metadata": self.metadata,
        }


class BootstrapAggregator:
    """
    Bootstrap resampling for uncertainty quantification in aggregation.

    Bootstrap is a non-parametric method that works for any aggregation function,
    including non-linear operations like Choquet integrals.
    """

    def __init__(self, n_samples: int = 1000, random_seed: int = 42):
        """
```

```
        Initialize bootstrap aggregator.

        Args:
            n_samples: Number of bootstrap resamples
            random_seed: Fixed seed for reproducibility
        """
        self.n_samples = n_samples
        self.rng = np.random.RandomState(random_seed)
        logger.info(f"BootstrapAggregator initialized (n_samples={n_samples}, seed={random_seed})")

    def bootstrap_weighted_average(
        self,
        scores: list[float],
        weights: list[float] | None = None,
    ) -> UncertaintyMetrics:
        """
        Compute weighted average with bootstrap confidence intervals.

        Args:
            scores: List of input scores
            weights: Optional weights (default: uniform)

        Returns:
            UncertaintyMetrics with full uncertainty decomposition

        Raises:
            ValueError: If scores is empty or weights mismatch
        """
        if not scores:
            raise ValueError("Cannot bootstrap empty score list")

        scores_arr = np.array(scores, dtype=np.float64)
        n = len(scores_arr)

        if weights is None:
            weights_arr = np.ones(n) / n
        else:
            if len(weights) != n:
                raise ValueError(f"Weight count {len(weights)} != score count {n}")
            weights_arr = np.array(weights, dtype=np.float64)
            weights_arr = weights_arr / np.sum(weights_arr)  # Normalize

        # Bootstrap resampling
        resamples = np.zeros(self.n_samples)
        for i in range(self.n_samples):
            # Sample with replacement
            indices = self.rng.choice(n, size=n, replace=True)
            resample_scores = scores_arr[indices]
            resample_weights = weights_arr[indices]
            resample_weights = resample_weights / np.sum(resample_weights)

            resamples[i] = np.sum(resample_scores * resample_weights)

        # Compute statistics
        mean = np.mean(resamples)
        std = np.std(resamples, ddof=1)
```

```
12798:         variance = std ** 2
12799:
12800:         # Confidence intervals (percentile method)
12801:         ci_95_lower = np.percentile(resamples, 2.5)
12802:         ci_95_upper = np.percentile(resamples, 97.5)
12803:         ci_99_lower = np.percentile(resamples, 0.5)
12804:         ci_99_upper = np.percentile(resamples, 99.5)
12805:
12806:         # Uncertainty decomposition
12807:         # Epistemic: Uncertainty in the aggregation (reducible with more samples)
12808:         epistemic_uncertainty = std
12809:
12810:         # Aleatoric: Inherent variability in data (irreducible)
12811:         aleatoric_uncertainty = np.std(scores_arr, ddof=1)
12812:
12813:         # Coefficient of variation
12814:         cv = std / mean if mean != 0.0 else float('inf')
12815:
12816:         # Distribution shape
12817:         skewness = stats.skew(resamples)
12818:         kurtosis = stats.kurtosis(resamples)
12819:
12820:         logger.debug(
12821:             f"Bootstrap complete: mean={mean:.4f}, std={std:.4f}, "
12822:             f"CI95=[{ci_95_lower:.4f}, {ci_95_upper:.4f}]"
12823:         )
12824:
12825:         return UncertaintyMetrics(
12826:             mean=float(mean),
12827:             std=float(std),
12828:             variance=float(variance),
12829:             confidence_interval_95=(float(ci_95_lower), float(ci_95_upper)),
12830:             confidence_interval_99=(float(ci_99_lower), float(ci_99_upper)),
12831:             epistemic_uncertainty=float(epistemic_uncertainty),
12832:             aleatoric_uncertainty=float(aleatoric_uncertainty),
12833:             coefficient_of_variation=float(cv),
12834:             skewness=float(skewness),
12835:             kurtosis=float(kurtosis),
12836:             metadata={
12837:                 "n_scores": n,
12838:                 "n_bootstrap_samples": self.n_samples,
12839:                 "original_scores_mean": float(np.mean(scores_arr)),
12840:                 "original_scores_std": float(np.std(scores_arr, ddof=1)),
12841:             },
12842:         )
12843:
12844:
12845: class BayesianPropagation:
12846:     """
12847:     Bayesian error propagation for analytical uncertainty quantification.
12848:
12849:     For linear aggregations (weighted averages), we can compute uncertainty
12850:     analytically without Monte Carlo sampling.
12851:     """
12852:
12853:     @staticmethod
```

```
    def propagate_weighted_average(
        scores: list[float],
        score_uncertainties: list[float],
        weights: list[float] | None = None,
    ) -> tuple[float, float]:
        """
        Analytical uncertainty propagation for weighted average.

        Formula:
            If Y = Î£ w_i * X_i, then
            Var(Y) = Î£ (w_i^2 * Var(X_i))  [assuming independence]

        Args:
            scores: List of score means
            score_uncertainties: List of score standard deviations
            weights: Optional weights (default: uniform)

        Returns:
            Tuple of (mean, std) for aggregated score

        Raises:
            ValueError: If input lists have mismatched lengths
        """
        n = len(scores)
        if len(score_uncertainties) != n:
            raise ValueError(
                f"Score count {n} != uncertainty count {len(score_uncertainties)}"
            )

        scores_arr = np.array(scores, dtype=np.float64)
        uncertainties_arr = np.array(score_uncertainties, dtype=np.float64)

        if weights is None:
            weights_arr = np.ones(n) / n
        else:
            if len(weights) != n:
                raise ValueError(f"Weight count {len(weights)} != score count {n}")
            weights_arr = np.array(weights, dtype=np.float64)
            weights_arr = weights_arr / np.sum(weights_arr)

        # Mean propagation
        mean = np.sum(weights_arr * scores_arr)

        # Variance propagation (assuming independence)
        variance = np.sum((weights_arr ** 2) * (uncertainties_arr ** 2))
        std = np.sqrt(variance)

        logger.debug(
            f"Bayesian propagation: mean={mean:.4f}, std={std:.4f} "
            f"(from {n} inputs)"
        )

        return float(mean), float(std)


class SensitivityAnalysis:
```

```
    """
    Sensitivity analysis for identifying influential inputs.

    Implements variance-based sensitivity analysis (Sobol indices) to quantify
    the contribution of each input to output variance.
    """

    @staticmethod
    def compute_sobol_indices(
        scores: list[float],
        weights: list[float],
        aggregation_func: callable = None,
        n_samples: int = 1000,
        random_seed: int = 42,
    ) -> dict[int, float]:
        """
        Compute first-order Sobol indices for weighted average.

        Sobol index S_i measures the fraction of output variance due to input i.

        Args:
            scores: Input scores
            weights: Input weights
            aggregation_func: Aggregation function (default: weighted average)
            n_samples: Monte Carlo samples for estimation
            random_seed: Reproducibility seed

        Returns:
            Dictionary mapping input index to Sobol index

        Note:
            For weighted average, Sobol index is proportional to weight^2 * var(score_i).
        """
        if aggregation_func is None:
            # Default: weighted average
            def aggregation_func(s, w):
                return np.sum(np.array(s) * np.array(w))

        n = len(scores)
        scores_arr = np.array(scores, dtype=np.float64)
        weights_arr = np.array(weights, dtype=np.float64)
        weights_arr = weights_arr / np.sum(weights_arr)

        rng = np.random.RandomState(random_seed)

        # Baseline output variance
        baseline_output = aggregation_func(scores_arr, weights_arr)

        # Perturb each input and measure output variance
        sobol_indices = {}
        for i in range(n):
            # Monte Carlo: sample score_i from distribution, hold others fixed
            perturbed_outputs = []
            for _ in range(n_samples):
                perturbed_scores = scores_arr.copy()
                # Perturb score_i (assume Â±20% noise)
```

```
                noise = rng.normal(0, 0.2 * abs(scores_arr[i]))
                perturbed_scores[i] = scores_arr[i] + noise

                perturbed_output = aggregation_func(perturbed_scores, weights_arr)
                perturbed_outputs.append(perturbed_output)

            # Sobol index = Var(E[Y|X_i]) / Var(Y)
            # Approximated by: Var of perturbed outputs
            variance_due_to_i = np.var(perturbed_outputs, ddof=1)
            sobol_indices[i] = float(variance_due_to_i)

        # Normalize so sum = 1.0
        total_variance = sum(sobol_indices.values())
        if total_variance > 0:
            sobol_indices = {k: v / total_variance for k, v in sobol_indices.items()}

        logger.debug(f"Sobol indices: {sobol_indices}")
        return sobol_indices


def aggregate_with_uncertainty(
    scores: list[float],
    weights: list[float] | None = None,
    n_bootstrap: int = 1000,
    random_seed: int = 42,
) -> tuple[float, UncertaintyMetrics]:
    """
    Convenience function: Aggregate scores with full uncertainty quantification.

    Args:
        scores: Input scores
        weights: Optional weights (default: uniform)
        n_bootstrap: Number of bootstrap samples
        random_seed: Reproducibility seed

    Returns:
        Tuple of (point_estimate, uncertainty_metrics)
    """
    bootstrapper = BootstrapAggregator(n_samples=n_bootstrap, random_seed=random_seed)
    uncertainty = bootstrapper.bootstrap_weighted_average(scores, weights)

    # Point estimate: use mean from bootstrap
    point_estimate = uncertainty.mean

    return point_estimate, uncertainty


__all__ = [
    "UncertaintyMetrics",
    "BootstrapAggregator",
    "BayesianPropagation",
    "SensitivityAnalysis",
    "aggregate_with_uncertainty",
]
```

```
13022:
13023: ================================================================================
13024: FILE: src/farfan_pipeline/question_context.py
13025: ================================================================================
13026:
13027: from __future__ import annotations
13028:
13029: from collections.abc import Mapping as ABCMapping
13030: from dataclasses import dataclass
13031: from types import MappingProxyType
13032: from typing import Any
13033:
13034:
13035: def _freeze(obj: Any) -> Any:  # noqa: ANN401  # type: ignore[misc]
13036:     if isinstance(obj, ABCMapping):
13037:         return MappingProxyType({k: _freeze(v) for k, v in obj.items()})
13038:     if isinstance(obj, list | tuple):
13039:         return tuple(_freeze(x) for x in obj)
13040:     if isinstance(obj, set):
13041:         return frozenset(_freeze(x) for x in obj)
13042:     return obj
13043:
13044:
13045: @dataclass(frozen=True, slots=True)
13046: class QuestionContext:
13047:     """Carries question requirements through entire pipeline (deep-immutable)."""
13048:
13049:     question_mapping: Any  # type: ignore[misc]
13050:     dnp_standards: ABCMapping[str, Any]  # type: ignore[misc]
13051:     required_evidence_types: tuple[str, ...]
13052:     search_queries: tuple[str, ...]
13053:     validation_criteria: ABCMapping[str, Any]  # type: ignore[misc]
13054:     traceability_id: str
13055:
13056:     def __post_init__(self) -> None:  # type: ignore[misc]
13057:         object.__setattr__(self, "question_mapping", _freeze(self.question_mapping))
13058:         object.__setattr__(self, "dnp_standards", _freeze(self.dnp_standards))
13059:         object.__setattr__(
13060:             self, "required_evidence_types", tuple(self.required_evidence_types)
13061:         )
13062:         object.__setattr__(self, "search_queries", tuple(self.search_queries))
13063:         object.__setattr__(
13064:             self, "validation_criteria", _freeze(self.validation_criteria)
13065:         )
13066:
13067:
13068:
13069: ================================================================================
13070: FILE: src/farfan_pipeline/scoring/__init__.py
13071: ================================================================================
13072:
13073: """Scoring module package.
13074:
13075: This package provides backward compatibility for code that imports
13076: scoring classes from farfan_core.scoring.
13077: """
```

```
13078:
13079: from farfan_pipeline.analysis.scoring.scoring import (
13080:     EvidenceStructureError,
13081:     ModalityConfig,
13082:     ModalityValidationError,
13083:     QualityLevel,
13084:     ScoredResult,
13085:     ScoringError,
13086:     ScoringModality,
13087:     ScoringValidator,
13088:     apply_rounding,
13089:     apply_scoring,
13090:     clamp,
13091:     determine_quality_level,
13092:     score_type_a,
13093:     score_type_b,
13094:     score_type_c,
13095:     score_type_d,
13096:     score_type_e,
13097:     score_type_f,
13098: )
13099:
13100: __all__ = [
13101:     "EvidenceStructureError",
13102:     "ModalityConfig",
13103:     "ModalityValidationError",
13104:     "QualityLevel",
13105:     "ScoredResult",
13106:     "ScoringError",
13107:     "ScoringModality",
13108:     "ScoringValidator",
13109:     "apply_rounding",
13110:     "apply_scoring",
13111:     "clamp",
13112:     "determine_quality_level",
13113:     "score_type_a",
13114:     "score_type_b",
13115:     "score_type_c",
13116:     "score_type_d",
13117:     "score_type_e",
13118:     "score_type_f",
13119: ]
13120:
13121:
13122:
13123: ================================================================================
13124: FILE: src/farfan_pipeline/scoring/scoring.py
13125: ================================================================================
13126:
13127: """Scoring module - re-exports from analysis.scoring.scoring.
13128:
13129: This module provides backward compatibility for code that imports
13130: scoring classes from farfan_core.scoring.scoring.
13131: """
13132:
13133: from farfan_pipeline.analysis.scoring.scoring import (
```

```
    EvidenceStructureError,
    ModalityConfig,
    ModalityValidationError,
    QualityLevel,
    ScoredResult,
    ScoringError,
    ScoringModality,
    ScoringValidator,
    apply_rounding,
    apply_scoring,
    clamp,
    determine_quality_level,
    score_type_a,
    score_type_b,
    score_type_c,
    score_type_d,
    score_type_e,
    score_type_f,
)

__all__ = [
    "EvidenceStructureError",
    "ModalityConfig",
    "ModalityValidationError",
    "QualityLevel",
    "ScoredResult",
    "ScoringError",
    "ScoringModality",
    "ScoringValidator",
    "apply_rounding",
    "apply_scoring",
    "clamp",
    "determine_quality_level",
    "score_type_a",
    "score_type_b",
    "score_type_c",
    "score_type_d",
    "score_type_e",
    "score_type_f",
]


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

"""
Runtime-facing console entrypoints for the SAAAAAA package.

Modules in this package are designed to be executed via
``python -m farfan_core.scripts.<name>`` so that repository
scripts no longer need to mutate ``sys.path`` at runtime.
"""

from __future__ import annotations
```

```
13190:
13191: __all__: list[str] = []
13192:
13193:
13194:
13195: ================================================================================
13196: FILE: src/farfan_pipeline/synchronization/__init__.py
13197: ================================================================================
13198:
13199: from farfan_pipeline.synchronization.irrigation_synchronizer import ChunkMatrix
13200:
13201: __all__ = ["ChunkMatrix"]
13202:
13203:
```