```
    dimension_scores: list[DimensionScore]
    validation_passed: bool = True
    validation_details: dict[str, Any] = field(default_factory=dict)
    cluster_id: str | None = None  # Used for grouping into clusters

@dataclass
class ClusterScore:
    """Represents the aggregated score for a MESO cluster, based on its policy areas."""
    cluster_id: str
    cluster_name: str
    areas: list[str]
    score: float
    coherence: float  # Coherence metric for the scores within this cluster
    variance: float
    weakest_area: str | None
    area_scores: list[AreaScore]
    validation_passed: bool = True
    validation_details: dict[str, Any] = field(default_factory=dict)

@dataclass
class MacroScore:
    """Represents the final, holistic macro evaluation score for the entire system."""
    score: float
    quality_level: str
    cross_cutting_coherence: float  # Coherence across all clusters
    systemic_gaps: list[str]
    strategic_alignment: float
    cluster_scores: list[ClusterScore]
    validation_passed: bool = True
    validation_details: dict[str, Any] = field(default_factory=dict)

class AggregationError(Exception):
    """Base exception for aggregation errors."""
    pass

class ValidationError(AggregationError):
    """Raised when validation fails."""
    pass

class WeightValidationError(ValidationError):
    """Raised when weight validation fails."""
    pass

class ThresholdValidationError(ValidationError):
    """Raised when threshold validation fails."""
    pass

class HermeticityValidationError(ValidationError):
    """Raised when hermeticity validation fails."""
    pass

class CoverageError(AggregationError):
    """Raised when coverage requirements are not met."""
    pass

class DimensionAggregator:
    """
    Aggregates micro question scores into dimension scores.

    Responsibilities:
    - Aggregate 5 micro questions (Q1-Q5) per dimension
    - Validate weights sum to 1.0
    - Apply rubric thresholds
    - Ensure coverage (abort if insufficient)
    - Provide detailed logging
    """

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

        # 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

        logger.info("DimensionAggregator initialized")

        # 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("saaaaaa.processing.aggregation.DimensionAggregator.validate_dimens
ion_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("saaaaaa.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("saaaaaa.processing.aggregation.DimensionAggregator.validate_weights")
    def validate_weights(self, weights: list[float]) -> tuple[bool, str]:
        """
        Ensures that a list of weights sums to get_parameter_loader().get("saaaaaa.process
ing.aggregation.DimensionAggregator.validate_weights").get("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 - get_parameter_loader().get("saaaaaa.processing.aggregation.Dim
ensionAggregator.validate_weights").get("auto_param_L603_28", 1.0)) > tolerance:
            msg = f"Weight sum validation failed: sum={weight_sum:.6f}, expected=get_param
eter_loader().get("saaaaaa.processing.aggregation.DimensionAggregator.validate_weights").g
et("auto_param_L604_81", 1.0)"
            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 get_parameter_loader().get("saaaaaa.processing.aggregation.DimensionAgg
regator.validate_weights").get("auto_param_L665_19", 0.0)

        if weights is None:
            # Equal weights
            weights = [get_parameter_loader().get("saaaaaa.processing.aggregation.Dimensio
nAggregator.validate_weights").get("auto_param_L669_23", 1.0) / len(scores)] * 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 get_parameter_loader().get("saaaaaa.processing.aggregati
on.DimensionAggregator.validate_weights").get("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}"
        )

        return weighted_sum

    def apply_rubric_thresholds(
        self,
        score: float,
        thresholds: dict[str, float] | None = None
    ) -> str:
        """
        Apply rubric thresholds to determine quality level.

        Args:
            score: Aggregated score (0-3 range)
            thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO,
ACEPTABLE)
                        Each value should be a normalized threshold (0-1 range)

        Returns:
            Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
        """
        # Clamp score to valid range [0, 3]
        clamped_score = max(get_parameter_loader().get("saaaaaa.processing.aggregation.Dim
ensionAggregator.validate_weights").get("auto_param_L714_28", 0.0), min(3.0, score))

        # Normalize to 0-1 range
        normalized_score = clamped_score / 3.0

        # Use provided thresholds or defaults
        if thresholds:
            excellent_threshold = thresholds.get('EXCELENTE', get_parameter_loader().get("
saaaaaa.processing.aggregation.DimensionAggregator.validate_weights").get("auto_param_L721
_62", 0.85))
            good_threshold = thresholds.get('BUENO', get_parameter_loader().get("saaaaaa.p
rocessing.aggregation.DimensionAggregator.validate_weights").get("auto_param_L722_53",
0.70))
            acceptable_threshold = thresholds.get('ACEPTABLE', get_parameter_loader().get(
"saaaaaa.processing.aggregation.DimensionAggregator.validate_weights").get("auto_param_L72
3_63", 0.55))
        else:
            excellent_threshold = get_parameter_loader().get("saaaaaa.processing.aggregati
on.DimensionAggregator.validate_weights").get("excellent_threshold", 0.85) # Refactored
            good_threshold = get_parameter_loader().get("saaaaaa.processing.aggregation.Di
mensionAggregator.validate_weights").get("good_threshold", 0.7) # Refactored
            acceptable_threshold = get_parameter_loader().get("saaaaaa.processing.aggregat
ion.DimensionAggregator.validate_weights").get("acceptable_threshold", 0.55) # Refactored

        # Apply thresholds
        if normalized_score >= excellent_threshold:
            quality = "EXCELENTE"
        elif normalized_score >= good_threshold:
            quality = "BUENO"
        elif normalized_score >= acceptable_threshold:
            quality = "ACEPTABLE"
        else:
            quality = "INSUFICIENTE"

        logger.debug(
            f"Rubric applied: score={score:.4f}, "
            f"normalized={normalized_score:.4f}, quality={quality}"
        )

        return quality

    def aggregate_dimension(
        self,
        scored_results: list[ScoredResult],
        group_by_values: dict[str, Any],
        weights: list[float] | None = None,
    ) -> DimensionScore:
        """
        Aggregate a single dimension from micro question results.

        Args:
            scored_results: List of scored results for this dimension/area.
            group_by_values: Dictionary of grouping keys and their values.
            weights: Optional weights for questions (defaults to equal weights).

        Returns:
            DimensionScore with aggregated score and quality level.

        Raises:
            ValidationError: If validation fails.
            CoverageError: If coverage is insufficient.
        """
        dimension_id = group_by_values.get("dimension", "UNKNOWN")
        area_id = group_by_values.get("policy_area", "UNKNOWN")
        logger.info(f"Aggregating dimension {dimension_id} for area {area_id}")

        validation_details = {}

        # In this context, scored_results are already grouped, so we can use them
directly.
        dim_results = scored_results

        expected_count = self._expected_question_count(area_id, dimension_id)
        # Validate coverage
        try:
            coverage_valid, coverage_msg = self.validate_coverage(
                dim_results,
                expected_count=expected_count or 5,
            )
            validation_details["coverage"] = {
                "valid": coverage_valid,
                "message": coverage_msg,
                "count": len(dim_results)
            }
        except CoverageError as e:
            logger.error(f"Coverage validation failed for {dimension_id}/{area_id}: {e}")
            # Return minimal score if aborted
            return DimensionScore(
                dimension_id=dimension_id,
                area_id=area_id,
                score=get_parameter_loader().get("saaaaaa.processing.aggregation.Dimension
Aggregator.validate_weights").get("auto_param_L795_22", 0.0),
                quality_level="INSUFICIENTE",
                contributing_questions=[],
                validation_passed=False,
                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=get_parameter_loader().get("saaaaaa.processing.aggregation.Dimension
Aggregator.validate_weights").get("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]

        # Calculate weighted average
        resolved_weights = weights or self._resolve_dimension_weights(dimension_id,
dim_results)
        try:
            avg_score = self.calculate_weighted_average(scores, resolved_weights)
            validation_details["weights"] = {
                "valid": True,
                "weights": resolved_weights if resolved_weights else "equal",
                "score": avg_score
            }
        except WeightValidationError as e:
            logger.error(f"Weight validation failed for {dimension_id}/{area_id}: {e}")
            return DimensionScore(
                dimension_id=dimension_id,
                area_id=area_id,
                score=get_parameter_loader().get("saaaaaa.processing.aggregation.Dimension
Aggregator.validate_weights").get("auto_param_L831_22", 0.0),
                quality_level="INSUFICIENTE",
                contributing_questions=[r.question_global for r in dim_results],
                validation_passed=False,
                validation_details={"error": str(e), "type": "weights"}
            )

        # Apply rubric thresholds
        quality_level = self.apply_rubric_thresholds(avg_score)
        validation_details["rubric"] = {
            "score": avg_score,
            "quality_level": quality_level
        }
        # Add score_max for downstream normalization
        validation_details["score_max"] = 3.0

        logger.info(
            f"✓ Dimension {dimension_id}/{area_id}: "
            f"score={avg_score:.4f}, quality={quality_level}"
        )

        return DimensionScore(
            dimension_id=dimension_id,
            area_id=area_id,
            score=avg_score,
            quality_level=quality_level,
            contributing_questions=[r.question_global for r in dim_results],
            validation_passed=True,
            validation_details=validation_details
        )

    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("saaaaaa.processing.aggregation.DimensionAggregator._expected_quest
ion_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
            weight = mapping.get(slot)
            if weight is None:
                logger.debug(
                    "Missing weight for slot %s in dimension %s – falling back to equal
weights",
                    slot,
                    dimension_id,
                )
                return None
            weights.append(weight)

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

def run_aggregation_pipeline(
    scored_results: list[dict[str, Any]],
    monolith: dict[str, Any],
    abort_on_insufficient: bool = True
) -> list[ClusterScore]:
    """
    Orchestrates the end-to-end aggregation pipeline.

    This function provides a high-level entry point to the aggregation system,
    demonstrating the sequential wiring of the aggregator components. It ensures
    that data flows from raw scored results through dimension, area, and
    finally cluster aggregation in a controlled and validated manner.

    Note on Parallelization: This implementation is sequential. For very large
    datasets, the `group_by` operations in each aggregator's `run` method
    could be parallelized (e.g., using `concurrent.futures`) to process
    independent groups concurrently.

    Args:
        scored_results: A list of dictionaries, each representing a raw scored result.
        monolith: The central monolith configuration object.
        abort_on_insufficient: If True, the pipeline will stop on validation errors.

    Returns:
        A list of aggregated ClusterScore objects.
    """
    # 1. Input Validation (Pre-flight check)
    validated_scored_results = validate_scored_results(scored_results)

    aggregation_settings = AggregationSettings.from_monolith(monolith)

    # 2. FASE 4: Dimension Aggregation
    dim_aggregator = DimensionAggregator(
        monolith,
        abort_on_insufficient,
        aggregation_settings=aggregation_settings,
    )
    dimension_scores = dim_aggregator.run(
        validated_scored_results,
        group_by_keys=dim_aggregator.dimension_group_by_keys,
    )

    # 3. FASE 5: Area Policy Aggregation
    area_aggregator = AreaPolicyAggregator(
        monolith,
        abort_on_insufficient,
        aggregation_settings=aggregation_settings,
    )
    area_scores = area_aggregator.run(
        dimension_scores,
        group_by_keys=area_aggregator.area_group_by_keys,
    )
    # 4. FASE 6: Cluster Aggregation
    cluster_aggregator = ClusterAggregator(
        monolith,
        abort_on_insufficient,
        aggregation_settings=aggregation_settings,
    )
    cluster_definitions = monolith["blocks"]["niveles_abstraccion"]["clusters"]
    cluster_scores = cluster_aggregator.run(
        area_scores,
        cluster_definitions
    )

    return cluster_scores

    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

class AreaPolicyAggregator:
    """
    Aggregates dimension scores into policy area scores.

    Responsibilities:
    - Aggregate 6 dimension scores per policy area
    - Validate dimension completeness
    - Apply area-level rubric thresholds
    - Ensure hermeticity (no dimension overlap)
    """

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

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

        logger.info("AreaPolicyAggregator initialized")

    def validate_hermeticity(
        self,
        dimension_scores: list[DimensionScore],
        area_id: str
    ) -> tuple[bool, str]:
        """
        Validate hermeticity (no dimension overlap/gaps).
        Uses scoped validation based on policy_area.dimension_ids from monolith.

        Args:
            dimension_scores: List of dimension scores for the area
            area_id: Policy area ID

        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}")
        return True, "Hermeticity validated"


@calibrated_method("saaaaaa.processing.aggregation.AreaPolicyAggregator.normalize_scores")
    def normalize_scores(self, dimension_scores: list[DimensionScore]) -> list[float]:
        """
        Normalize dimension scores to 0-1 range.

        Args:
            dimension_scores: List of dimension scores

        Returns:
            List of normalized scores
        """
        normalized = []
        for d in dimension_scores:
            # Extract max_expected from validation_details or default to 3.0
            max_expected = d.validation_details.get('score_max', 3.0) if
d.validation_details else 3.0
            normalized.append(max(get_parameter_loader().get("saaaaaa.processing.aggregati
on.AreaPolicyAggregator.normalize_scores").get("auto_param_L1148_34", 0.0),
min(max_expected, d.score)) / max_expected)

        logger.debug(f"Scores normalized: {normalized}")
        return normalized

    def apply_rubric_thresholds(
        self,
        score: float,
        thresholds: dict[str, float] | None = None
    ) -> str:
        """
        Apply area-level rubric thresholds.

        Args:
            score: Aggregated score (0-3 range)
            thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO,
ACEPTABLE)
                       Each value should be a normalized threshold (0-1 range)

        Returns:
            Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
        """
        # Clamp score to valid range [0, 3]
        clamped_score = max(get_parameter_loader().get("saaaaaa.processing.aggregation.Are
aPolicyAggregator.normalize_scores").get("auto_param_L1170_28", 0.0), min(3.0, score))
```

```
        # Normalize to 0-1 range
        normalized_score = clamped_score / 3.0

        # Use provided thresholds or defaults
        if thresholds:
            excellent_threshold = thresholds.get('EXCELENTE', get_parameter_loader().get("
saaaaaa.processing.aggregation.AreaPolicyAggregator.normalize_scores").get("auto_param_L11
77_62", 0.85))
            good_threshold = thresholds.get('BUENO', get_parameter_loader().get("saaaaaa.p
rocessing.aggregation.AreaPolicyAggregator.normalize_scores").get("auto_param_L1178_53",
0.70))
            acceptable_threshold = thresholds.get('ACEPTABLE', get_parameter_loader().get(
"saaaaaa.processing.aggregation.AreaPolicyAggregator.normalize_scores").get("auto_param_L1
179_63", 0.55))
        else:
            excellent_threshold = get_parameter_loader().get("saaaaaa.processing.aggregati
on.AreaPolicyAggregator.normalize_scores").get("excellent_threshold", 0.85) # Refactored
            good_threshold = get_parameter_loader().get("saaaaaa.processing.aggregation.Ar
eaPolicyAggregator.normalize_scores").get("good_threshold", 0.7) # Refactored
            acceptable_threshold = get_parameter_loader().get("saaaaaa.processing.aggregat
ion.AreaPolicyAggregator.normalize_scores").get("acceptable_threshold", 0.55) # Refactored

        # Apply thresholds
        if normalized_score >= excellent_threshold:
            quality = "EXCELENTE"
        elif normalized_score >= good_threshold:
            quality = "BUENO"
        elif normalized_score >= acceptable_threshold:
            quality = "ACEPTABLE"
        else:
            quality = "INSUFICIENTE"

        logger.debug(
            f"Area rubric applied: score={score:.4f}, "
            f"normalized={normalized_score:.4f}, quality={quality}"
        )

        return quality

    def aggregate_area(
        self,
        dimension_scores: list[DimensionScore],
        group_by_values: dict[str, Any],
        weights: list[float] | None = None,
    ) -> AreaScore:
        """
        Aggregate a single policy area from dimension scores.

        Args:
            dimension_scores: List of dimension scores for this area.
            group_by_values: Dictionary of grouping keys and their values.
            weights: Optional list of weights for dimension scores.

        Returns:
            AreaScore with aggregated score and quality level.

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

        validation_details = {}

        # The dimension_scores are already grouped.
        area_dim_scores = dimension_scores
        # Validate hermeticity
        try:
            hermetic_valid, hermetic_msg = self.validate_hermeticity(area_dim_scores,
area_id)
            validation_details["hermeticity"] = {
                "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=get_parameter_loader().get("saaaaaa.processing.aggregation.AreaPolic
yAggregator.normalize_scores").get("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=get_parameter_loader().get("saaaaaa.processing.aggregation.AreaPolic
yAggregator.normalize_scores").get("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
        }

        # 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
        )

        logger.info(
            f"✓ Policy area {area_id} ({area_name}): "
            f"score={avg_score:.4f}, quality={quality_level}"
        )

        return AreaScore(
            area_id=area_id,
            area_name=area_name,
            score=avg_score,
            quality_level=quality_level,
            dimension_scores=area_dim_scores,
            validation_passed=True,
            validation_details=validation_details
        )

    def run(
        self,
        dimension_scores: list[DimensionScore],
        group_by_keys: list[str]
    ) -> list[AreaScore]:
        """
        Run the area aggregation process.

        Args:
            dimension_scores: List of all dimension scores.
            group_by_keys: List of keys to group by.

        Returns:
            A list of AreaScore objects.
        """
        def key_func(d):
            return tuple(getattr(d, key) for key in group_by_keys)
        grouped_scores = group_by(dimension_scores, key_func)

        area_scores = []
        for group_key, scores in grouped_scores.items():
            group_by_values = dict(zip(group_by_keys, group_key, strict=False))
            score = self.aggregate_area(scores, group_by_values, weights=None)
            area_scores.append(score)

        return area_scores

    def _resolve_area_weights(
        self,
        area_id: str,
        dimension_scores: list[DimensionScore],
    ) -> list[float] | None:
        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 – 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 = get_parameter_loader().get("saaaaaa.processing.aggregation.AreaPolicy
Aggregator.normalize_scores").get("PENALTY_WEIGHT", 0.3) # Refactored
    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
        if monolith is not None:
            self.scoring_config = monolith["blocks"]["scoring"]
            self.niveles = monolith["blocks"]["niveles_abstraccion"]
            self.clusters = self.niveles["clusters"]
        else:
            self.scoring_config = None
            self.niveles = None
            self.clusters = None

        logger.info("ClusterAggregator initialized")

    def validate_cluster_hermeticity(
        self,
        cluster_def: dict[str, Any],
        area_scores: list[AreaScore]
    ) -> tuple[bool, str]:
        """
        Validate cluster hermeticity.

        Args:
            cluster_def: Cluster definition from monolith
            area_scores: List of area scores for this cluster
        Returns:
            Tuple of (is_valid, message)

        Raises:
            HermeticityValidationError: If hermeticity is violated
        """
        expected_areas = cluster_def.get("policy_area_ids", [])
        actual_areas = [a.area_id for a in area_scores]

        # Check for duplicate areas
        if len(actual_areas) != len(set(actual_areas)):
            msg = (
                f"Cluster hermeticity violation: "
                f"duplicate areas found for cluster {cluster_def['cluster_id']}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        # Check that all expected areas are present
        missing_areas = set(expected_areas) - set(actual_areas)
        if missing_areas:
            msg = (
                f"Cluster hermeticity violation: "
                f"missing areas {missing_areas} for cluster {cluster_def['cluster_id']}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        # Check for unexpected areas
        extra_areas = set(actual_areas) - set(expected_areas)
        if extra_areas:
            msg = (
                f"Cluster hermeticity violation: "
                f"unexpected areas {extra_areas} for cluster {cluster_def['cluster_id']}"
            )
            logger.error(msg)
            if self.abort_on_insufficient:
                raise HermeticityValidationError(msg)
            return False, msg

        logger.debug(f"Cluster hermeticity validated for {cluster_def['cluster_id']}")
        return True, "Cluster hermeticity validated"

    def apply_cluster_weights(
        self,
        area_scores: list[AreaScore],
        weights: list[float] | None = None
    ) -> float:
        """
        Apply cluster-specific weights to area scores.

        Args:
            area_scores: List of area scores
            weights: Optional weights (defaults to equal weights)

        Returns:
            Weighted average score

        Raises:
            WeightValidationError: If weights validation fails
        """
        scores = [a.score for a in area_scores]

        if weights is None:
            # Equal weights
            weights = [get_parameter_loader().get("saaaaaa.processing.aggregation.AreaPoli
cyAggregator.normalize_scores").get("auto_param_L1495_23", 1.0) / len(scores)] *
len(scores)

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

        # Validate weights sum to get_parameter_loader().get("saaaaaa.processing.aggregati
on.AreaPolicyAggregator.normalize_scores").get("auto_param_L1507_34", 1.0)
        weight_sum = sum(weights)
        tolerance = 1e-6
        if abs(weight_sum - get_parameter_loader().get("saaaaaa.processing.aggregation.Are
aPolicyAggregator.normalize_scores").get("auto_param_L1510_28", 1.0)) > tolerance:
            msg = f"Cluster weight validation failed: sum={weight_sum:.6f}"
            logger.error(msg)
            if self.abort_on_insufficient:
                raise WeightValidationError(msg)

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

        logger.debug(
            f"Cluster weights applied: scores={scores}, "
            f"weights={weights}, result={weighted_avg:.4f}"
        )

        return weighted_avg


@calibrated_method("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence")
    def analyze_coherence(self, area_scores: list[AreaScore]) -> float:
        """
        Analyze cluster coherence.

        Coherence is measured as the inverse of standard deviation.
        Higher coherence means scores are more consistent.

        Args:
            area_scores: List of area scores

        Returns:
            Coherence value (0-1, where 1 is perfect coherence)
        """
        scores = [a.score for a in area_scores]

        if len(scores) <= 1:
            return get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggre
gator.analyze_coherence").get("auto_param_L1543_19", 1.0)

        # Calculate mean
        mean = sum(scores) / len(scores)

        # Calculate standard deviation
        variance = sum((s - mean) ** 2 for s in scores) / len(scores)
        std_dev = variance ** get_parameter_loader().get("saaaaaa.processing.aggregation.C
lusterAggregator.analyze_coherence").get("auto_param_L1550_30", 0.5)

        # Convert to coherence (inverse relationship)
        # Normalize by max possible std dev (3.0 for 0-3 range)
        max_std = 3.0
        coherence = max(get_parameter_loader().get("saaaaaa.processing.aggregation.Cluster
Aggregator.analyze_coherence").get("auto_param_L1555_24", 0.0), get_parameter_loader().get
("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L15
55_29", 1.0) - (std_dev / max_std))

        logger.debug(
            f"Coherence analysis: mean={mean:.4f}, "
            f"std_dev={std_dev:.4f}, coherence={coherence:.4f}"
        )

        return coherence

    def aggregate_cluster(
        self,
        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=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAg
gregator.analyze_coherence").get("auto_param_L1600_22", 0.0),
                coherence=get_parameter_loader().get("saaaaaa.processing.aggregation.Clust
erAggregator.analyze_coherence").get("auto_param_L1601_26", 0.0),
                variance=get_parameter_loader().get("saaaaaa.processing.aggregation.Cluste
rAggregator.analyze_coherence").get("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=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAg
gregator.analyze_coherence").get("auto_param_L1631_22", 0.0),
                coherence=get_parameter_loader().get("saaaaaa.processing.aggregation.Clust
erAggregator.analyze_coherence").get("auto_param_L1632_26", 0.0),
                variance=get_parameter_loader().get("saaaaaa.processing.aggregation.Cluste
rAggregator.analyze_coherence").get("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=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAg
gregator.analyze_coherence").get("auto_param_L1646_22", 0.0),
                coherence=get_parameter_loader().get("saaaaaa.processing.aggregation.Clust
erAggregator.analyze_coherence").get("auto_param_L1647_26", 0.0),
                variance=get_parameter_loader().get("saaaaaa.processing.aggregation.Cluste
rAggregator.analyze_coherence").get("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=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAg
gregator.analyze_coherence").get("auto_param_L1670_22", 0.0),
                coherence=get_parameter_loader().get("saaaaaa.processing.aggregation.Clust
erAggregator.analyze_coherence").get("auto_param_L1671_26", 0.0),
                variance=get_parameter_loader().get("saaaaaa.processing.aggregation.Cluste
rAggregator.analyze_coherence").get("auto_param_L1672_25", 0.0),
                weakest_area=None,
                area_scores=cluster_area_scores,
                validation_passed=False,
                validation_details={"error": str(e), "type": "weights"}
            )

        # Analyze coherence and variance metrics
        coherence = self.analyze_coherence(cluster_area_scores)
        scores_array = [a.score for a in cluster_area_scores]
        if scores_array:
            mean_score = sum(scores_array) / len(scores_array)
            variance = sum((score - mean_score) ** 2 for score in scores_array) /
len(scores_array)
        else:
            variance = get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterA
ggregator.analyze_coherence").get("variance", 0.0) # Refactored
        weakest_area = min(cluster_area_scores, key=lambda a: a.score, default=None)

        std_dev = variance ** get_parameter_loader().get("saaaaaa.processing.aggregation.C
lusterAggregator.analyze_coherence").get("auto_param_L1689_30", 0.5)
        normalized_std = min(std_dev / self.MAX_SCORE, get_parameter_loader().get("saaaaaa
.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1690_55",
1.0)) if std_dev > 0 else get_parameter_loader().get("saaaaaa.processing.aggregation.Clust
erAggregator.analyze_coherence").get("auto_param_L1690_80", 0.0)
        penalty_factor = get_parameter_loader().get("saaaaaa.processing.aggregation.Cluste
rAggregator.analyze_coherence").get("auto_param_L1691_25", 1.0) - (normalized_std *
self.PENALTY_WEIGHT)
        adjusted_score = weighted_score * penalty_factor

        validation_details["coherence"] = {
            "value": coherence,
            "interpretation": "high" if coherence > get_parameter_loader().get("saaaaaa.pr
ocessing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1696_52", 0.8)
 else "medium" if coherence > get_parameter_loader().get("saaaaaa.processing.aggregation.C
lusterAggregator.analyze_coherence").get("auto_param_L1696_85", 0.6) else "low"
        }
        validation_details["variance"] = variance
        if weakest_area:
            validation_details["weakest_area"] = weakest_area.area_id
        validation_details["imbalance_penalty"] = {
            "std_dev": std_dev,
            "penalty_factor": penalty_factor,
            "raw_score": weighted_score,
            "adjusted_score": adjusted_score,
        }

        logger.info(
            f"✓ Cluster {cluster_id} ({cluster_name}): "
            f"score={adjusted_score:.4f}, coherence={coherence:.4f}"
        )

        return ClusterScore(
            cluster_id=cluster_id,
            cluster_name=cluster_name,
            areas=expected_areas,
            score=adjusted_score,
            coherence=coherence,
            variance=variance,
            weakest_area=weakest_area.area_id if weakest_area else None,
            area_scores=cluster_area_scores,
            validation_passed=True,
            validation_details=validation_details
        )

    def run(
        self,
        area_scores: list[AreaScore],
        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 – falling back to equal
weights",
                    area_score.area_id,
                    cluster_id,
                )
                return None
            weights.append(weight)

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

class MacroAggregator:
    """
    Performs holistic macro evaluation (Q305).

    Responsibilities:
    - Aggregate all cluster scores
    - Calculate cross-cutting coherence
    - Identify systemic gaps
    - Assess strategic alignment
    """

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

        # 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

        logger.info("MacroAggregator initialized")

    def calculate_cross_cutting_coherence(
        self,
        cluster_scores: list[ClusterScore]
    ) -> float:
        """
        Calculate cross-cutting coherence across all clusters.

        Args:
            cluster_scores: List of cluster scores

        Returns:
            Cross-cutting coherence value (0-1)
        """
        scores = [c.score for c in cluster_scores]

        if len(scores) <= 1:
            return get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggre
gator.analyze_coherence").get("auto_param_L1847_19", 1.0)

        # Calculate mean
        mean = sum(scores) / len(scores)

        # Calculate standard deviation
        variance = sum((s - mean) ** 2 for s in scores) / len(scores)
        std_dev = variance ** get_parameter_loader().get("saaaaaa.processing.aggregation.C
lusterAggregator.analyze_coherence").get("auto_param_L1854_30", 0.5)

        # Convert to coherence
        max_std = 3.0
        coherence = max(get_parameter_loader().get("saaaaaa.processing.aggregation.Cluster
Aggregator.analyze_coherence").get("auto_param_L1858_24", 0.0), get_parameter_loader().get
("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L18
58_29", 1.0) - (std_dev / max_std))

        logger.debug(
            f"Cross-cutting coherence: mean={mean:.4f}, "
            f"std_dev={std_dev:.4f}, coherence={coherence:.4f}"
        )

        return coherence

    def identify_systemic_gaps(
        self,
        area_scores: list[AreaScore]
    ) -> list[str]:
        """
        Identify systemic gaps (areas with INSUFICIENTE quality).

        Args:
            area_scores: List of area scores

        Returns:
            List of area names with systemic gaps
        """
        gaps = []
        for area in area_scores:
            if area.quality_level == "INSUFICIENTE":
                gaps.append(area.area_name)
                logger.warning(f"Systemic gap identified: {area.area_name}")

        logger.info(f"Systemic gaps identified: {len(gaps)}")
        return gaps

    def assess_strategic_alignment(
        self,
        cluster_scores: list[ClusterScore],
        dimension_scores: list[DimensionScore]
    ) -> float:
        """
        Assess strategic alignment across all levels.

        Args:
            cluster_scores: List of cluster scores
            dimension_scores: List of dimension scores

        Returns:
            Strategic alignment score (0-1)
        """
        # Calculate average cluster coherence
        cluster_coherence = (
            sum(c.coherence for c in cluster_scores) / len(cluster_scores)
            if cluster_scores else get_parameter_loader().get("saaaaaa.processing.aggregat
ion.ClusterAggregator.analyze_coherence").get("auto_param_L1907_35", 0.0)
        )

        # Calculate dimension validation rate
        validated_dims = sum(1 for d in dimension_scores if d.validation_passed)
        validation_rate = validated_dims / len(dimension_scores) if dimension_scores else
get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coher
ence").get("auto_param_L1912_90", 0.0)

        # Strategic alignment is weighted combination
        alignment = (get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAgg
regator.analyze_coherence").get("auto_param_L1915_21", 0.6) * cluster_coherence) + (get_pa
rameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence")
.get("auto_param_L1915_49", 0.4) * validation_rate)

        logger.debug(
            f"Strategic alignment: cluster_coherence={cluster_coherence:.4f}, "
            f"validation_rate={validation_rate:.4f}, alignment={alignment:.4f}"
        )

        return alignment

    def apply_rubric_thresholds(
        self,
        score: float,
        thresholds: dict[str, float] | None = None
    ) -> str:
        """
        Apply macro-level rubric thresholds.

        Args:
            score: Aggregated macro score (0-3 range)
            thresholds: Optional threshold definitions (dict with keys: EXCELENTE, BUENO, ACEPTABLE)
                        Each value should be a normalized threshold (0-1 range)

        Returns:
            Quality level (EXCELENTE, BUENO, ACEPTABLE, INSUFICIENTE)
        """
        # Clamp score to valid range [0, 3]
        clamped_score = max(get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1941_28", 0.0), min(3.0, score))

        # Normalize to 0-1 range
        normalized_score = clamped_score / 3.0

        # Use provided thresholds or defaults
        if thresholds:
            excellent_threshold = thresholds.get('EXCELENTE', get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1948_62", 0.85))
            good_threshold = thresholds.get('BUENO', get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1949_53", 0.70))
            acceptable_threshold = thresholds.get('ACEPTABLE', get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1950_63", 0.55))
        else:
            excellent_threshold = get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("excellent_threshold", 0.85) # Refactored
            good_threshold = get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("good_threshold", 0.7) # Refactored
            acceptable_threshold = get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("acceptable_threshold", 0.55) # Refactored

        # Apply thresholds
        if normalized_score >= excellent_threshold:
            quality = "EXCELENTE"
        elif normalized_score >= good_threshold:
            quality = "BUENO"
        elif normalized_score >= acceptable_threshold:
            quality = "ACEPTABLE"
        else:
            quality = "INSUFICIENTE"

        logger.debug(
            f"Macro rubric applied: score={score:.4f}, "
            f"normalized={normalized_score:.4f}, quality={quality}"
        )

        return quality

    def evaluate_macro(
        self,
        cluster_scores: list[ClusterScore],
        area_scores: list[AreaScore],
        dimension_scores: list[DimensionScore]
    ) -> MacroScore:
        """
        Perform holistic macro evaluation (Q305).

        Args:
            cluster_scores: List of cluster scores (MESO level)
            area_scores: List of area scores
            dimension_scores: List of dimension scores

        Returns:
            MacroScore with holistic evaluation
        """
        logger.info("Performing macro holistic evaluation (Q305)")

        validation_details = {}

        if not cluster_scores:
            logger.error("No cluster scores available for macro evaluation")
            return MacroScore(
                score=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1997_22", 0.0),
                quality_level="INSUFICIENTE",
                cross_cutting_coherence=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("auto_param_L1999_40", 0.0),
                systemic_gaps=[],
                strategic_alignment=get_parameter_loader().get("saaaaaa.processing.aggregation.ClusterAggregator.analyze_coherence").get("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"✓ 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("saaaaaa.processing.aggregation.MacroAggregator._calculate_macro_sc
ore")
    def _calculate_macro_score(self, cluster_scores: list[ClusterScore]) -> float:
        weights = self.aggregation_settings.macro_cluster_weights
        if not cluster_scores:
            return get_parameter_loader().get("saaaaaa.processing.aggregation.MacroAggrega
tor._calculate_macro_score").get("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 – 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/saaaaaa/processing/cpp_ingestion/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 saaaaaa import get_parameter_loader
from saaaaaa.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:
    """Confidence scores for various extraction processes."""
    layout: float = 1.0
    ocr: float = 1.0
    typing: float = 1.0


@dataclass
class PolicyFacet:
    """Policy-related metadata facets."""
    programs: list[str] = field(default_factory=list)
    projects: list[str] = field(default_factory=list)
    axes: list[str] = field(default_factory=list)


@dataclass
class TimeFacet:
    """Temporal metadata facets."""
    years: list[int] = field(default_factory=list)
    periods: list[str] = field(default_factory=list)


@dataclass
class GeoFacet:
    """Geographic metadata facets."""
    territories: list[str] = field(default_factory=list)
    regions: list[str] = field(default_factory=list)


@dataclass
class ProvenanceMap:
    """Provenance information for chunk extraction."""
    source_page: int | None = None
    source_section: str | None = None
    extraction_method: str = "semantic_chunking"


@dataclass
class Budget:
    """Budget information extracted from policy document."""
    source: str
    use: str
    amount: float
    year: int
    currency: str = "COP"


# Alias for compatibility
BudgetInfo = Budget


@dataclass
class KPI:
    """Key Performance Indicator extracted from policy."""
    indicator_name: str
    target_value: float | None = None
    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("saaaaaa.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("saaaaaa.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))


@dataclass
class PolicyManifest:
    """
    High-level manifest summarizing policy structure.
    """
    axes: list[str] = field(default_factory=list)
    programs: list[str] = field(default_factory=list)
    projects: list[str] = field(default_factory=list)
    years: list[int] = field(default_factory=list)
    territories: list[str] = field(default_factory=list)
    indicators: list[str] = field(default_factory=list)
    budget_rows: int = 0


@dataclass
class QualityMetrics:
    """
    Quality metrics for the ingestion process.
    """
    boundary_f1: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L167_25", 0.0)
    kpi_linkage_rate: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L168_30", 0.0)
    budget_consistency_score: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L169_38", 0.0)
    provenance_completeness: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L170_37", 0.0)
    structural_consistency: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L171_36", 0.0)
    temporal_robustness: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L172_33", 0.0)
    chunk_context_coverage: float = get_parameter_loader().get("saaaaaa.processing.cpp_ingestion.models.ChunkGraph.add_edge").get("auto_param_L173_36", 0.0)


@dataclass
class IntegrityIndex:
    """
    Cryptographic integrity verification data.

    Uses BLAKE2b (not BLAKE3) for aggregate hash computation.
    Implementation uses hashlib.blake2b over JSON-serialized chunk hashes.
    """
    blake2b_root: str  # Aggregate hash (BLAKE2b-256) of all chunk hashes
    chunk_hashes: dict[str, str] = field(default_factory=dict)


@dataclass
class CanonPolicyPackage:
    """
    Canon Policy Package - Complete output from phase-one ingestion.

    This is the top-level container for all ingestion results.
    Also known as Smart Policy Chunks (SPC) in newer terminology.
    """
    schema_version: str
    chunk_graph: ChunkGraph

    # Optional high-level metadata
    policy_manifest: PolicyManifest | None = None
    quality_metrics: QualityMetrics | None = None
    integrity_index: IntegrityIndex | None = None

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

===== FILE: src/saaaaaa/processing/document_ingestion.py =====
"""
LEGACY INGESTION MODULE (DEPRECATED)

⚠  DEPRECATION WARNING ⚠
================================================================
This module implements a pre-SPC ingestion pipeline that produces
PreprocessedDocument directly from PDFs/text. It MUST NOT be used in the
canonical F.A.R.F.A.N pipeline.
```

```
Canonical Phase One path is:

    scripts/run_policy_pipeline_verified.py
      → saaaaaa.processing.spc_ingestion.CPPIngestionPipeline
      → saaaaaa.utils.spc_adapter.SPCAdapter
      → Orchestrator

The SPC (Smart Policy Chunks) system provides:
- 15-phase comprehensive analysis (causal, temporal, argumentative)
- 8 ChunkTypes with policy-aware classification
- 6 dimensions of Theory of Change analysis
- Quality gates and validation
- BGE-M3 embeddings and semantic chunking
- Complete provenance and integrity tracking

DO NOT import or use this module for any new code.
================================================================
    outcome = pipeline.ingest(input_path, output_dir)

    # Convert to PreprocessedDocument for orchestrator
    adapter = SPCAdapter()
    doc = adapter.to_preprocessed_document(outcome.cpp)

See: docs/CPP_ARCHITECTURE.md for complete documentation
See: examples/cpp_ingestion_example.py for usage examples

================================================================
Archivo: document_ingestion.py
Código: DI (LEGACY)
Propósito: Carga inicial de documentos PDF y extracción de texto

MÉTODOS (9 EXACTOS):
1. DocumentLoader.load_pdf()
2. DocumentLoader.validate_pdf()
3. DocumentLoader.extract_metadata()
4. TextExtractor.extract_full_text()
5. TextExtractor.extract_by_page()
6. TextExtractor.preserve_structure()
7. PreprocessingEngine.preprocess_document()
8. PreprocessingEngine.normalize_encoding()
9. PreprocessingEngine.detect_language()

INTEGRACIÓN CON MÓDULOS EXISTENTES:
- Usa PP.PolicyTextProcessor.normalize_unicode()
- Usa PP.PolicyTextProcessor.segment_into_sentences()
- Usa FV.PDETMunicipalPlanAnalyzer.extract_tables()
- Usa FV.PDETMunicipalPlanAnalyzer._clean_dataframe()
- Usa FV.PDETMunicipalPlanAnalyzer._classify_tables()

DEPENDENCIAS:
    pip install pdfplumber PyPDF2 spacy langdetect
    python -m spacy download es_core_news_sm
"""

import logging
import warnings
from collections.abc import Mapping, MutableMapping, Sequence
from dataclasses import dataclass
from datetime import datetime
from pathlib import Path
from types import MappingProxyType
from typing import Any, Optional

from saaaaaa.core.runtime_config import RuntimeConfig, get_runtime_config
from saaaaaa.core.contracts.runtime_contracts import (
    LanguageTier,
    LanguageDetectionInfo,
    FallbackCategory,
)
from saaaaaa.core.observability.structured_logging import log_fallback, get_logger
from saaaaaa.core.observability.metrics import increment_fallback

from saaaaaa.core.orchestrator.core import PreprocessedDocument
from dataclasses import dataclass, field
from typing import Any, List, Dict, Optional
from saaaaaa.core.calibration.decorators import calibrated_method

# Local definitions for missing schema classes (Legacy Support)
@dataclass
class SentenceMetadata:
    index: int
    page_number: int
    start_char: int
    end_char: int
    extra: Dict[str, Any] = field(default_factory=dict)

@dataclass
class TableAnnotation:
    table_id: str
    label: str
    attributes: Dict[str, Any]

@dataclass
class StructuredSection:
    title: str
    start_char: int
    content: str

@dataclass
class StructuredTextV1:
    full_text: str
    sections: List[StructuredSection]
    page_boundaries: List[Any]

@dataclass
class DocumentIndexesV1:
    term_index: Dict[str, Any]
    numeric_index: Dict[str, Any]
    temporal_index: Dict[str, Any]
    entity_index: Dict[str, Any]


# Issue deprecation warning when module is imported
warnings.warn(
    "document_ingestion module is deprecated. Use cpp_ingestion instead. "
    "See docs/CPP_ARCHITECTURE.md for migration guide.",
    DeprecationWarning,
    stacklevel=2
)

_EMPTY_MAPPING: Mapping[str, Any] = MappingProxyType({})

def _to_frozen_mapping(data: Mapping[str, Any] | None) -> Mapping[str, Any]:
    if not data:
        return _EMPTY_MAPPING
    if isinstance(data, MappingProxyType):
        return data
    return MappingProxyType(dict(data))

def _coerce_optional_int(value: Any) -> int | None:
    try:
        return int(value) if value is not None else None
    except (TypeError, ValueError):
        return None
```

```
def _build_sentence_metadata_entries(
    entries: Sequence[Any],
    sentences: Sequence[str],
) -> tuple[SentenceMetadata, ...]:
    result: list[SentenceMetadata] = []
    for index, entry in enumerate(entries):
        if isinstance(entry, SentenceMetadata):
            result.append(entry)
            continue

        if isinstance(entry, Mapping):
            metadata_dict = dict(entry)
            idx = int(metadata_dict.pop('index', index) or index)
            page_number = _coerce_optional_int(
                metadata_dict.pop('page', metadata_dict.pop('page_number', None))
            )
            start_char = _coerce_optional_int(metadata_dict.pop('start_char', None))
            end_char = _coerce_optional_int(metadata_dict.pop('end_char', None))
            result.append(
                SentenceMetadata(
                    index=idx,
                    page_number=page_number,
                    start_char=start_char,
                    end_char=end_char,
                    extra=_to_frozen_mapping(metadata_dict),
                )
            )
            continue

        result.append(SentenceMetadata(index=index))

    if not result:
        result = [SentenceMetadata(index=index) for index, _ in enumerate(sentences)]

    return tuple(result)

def _coerce_table_annotations(tables: Sequence[Any]) -> tuple[TableAnnotation, ...]:
    annotations: list[TableAnnotation] = []
    for index, table in enumerate(tables):
        if isinstance(table, TableAnnotation):
            annotations.append(table)
            continue

        if isinstance(table, Mapping):
            table_dict = dict(table)
            table_id = str(table_dict.pop('table_id', table_dict.pop('id',
f'table_{index}')))
            label = str(table_dict.pop('label', table_id))
            annotations.append(
                TableAnnotation(
                    table_id=table_id,
                    label=label,
                    attributes=_to_frozen_mapping(table_dict),
                )
            )
            continue

        annotations.append(
            TableAnnotation(
                table_id=f'table_{index}',
                label=type(table).__name__,
                attributes=_EMPTY_MAPPING,
            )
        )

    return tuple(annotations)

# PDF Processing
# Optional dependency - langdetect
try:
    from langdetect import LangDetectException, detect
    LANGDETECT_AVAILABLE = True
except ImportError:
    LANGDETECT_AVAILABLE = False
    # Dummy implementation
    class LangDetectException(Exception):
        pass
    def detect(text: str) -> str:
        return "es"  # Default to Spanish

# Optional dependency - PyPDF2
try:
    from PyPDF2 import PdfReader
    PYPDF2_AVAILABLE = True
except ImportError:
    PYPDF2_AVAILABLE = False
    PdfReader = None  # type: ignore

# Importar módulos existentes del sistema
# NOTA: Estos imports asumen la estructura existente del proyecto
try:
    from methods.financiero_viabilidad_tablas import PDETMunicipalPlanAnalyzer
    from methods.policy_processor import PolicyTextProcessor
except ImportError:
    # Fallback para testing standalone
    PolicyTextProcessor = None
    PDETMunicipalPlanAnalyzer = None

logger = logging.getLogger(__name__)

# ============================================================================
# DATACLASSES - ESTRUCTURAS DE DATOS INMUTABLES
# ============================================================================

@dataclass(frozen=True, slots=True)
class RawDocument:
    """
    Documento PDF crudo cargado desde disco.
    Inmutable para garantizar trazabilidad.
    """
    file_path: str
    file_name: str
    num_pages: int
    file_size_bytes: int
    file_hash: str
    metadata: Mapping[str, Any] = _EMPTY_MAPPING
    is_valid: bool = True

# ============================================================================
# CLASE 1: DocumentLoader
# ============================================================================

class DocumentLoader:
    """
    Carga y valida documentos PDF.
    Responsable de la I/O básica y validación inicial.
    """

    def __init__(self) -> None:
        self.logger = logger

    @calibrated_method("saaaaaa.processing.document_ingestion.DocumentLoader.load_pdf")
    def load_pdf(self, *, pdf_path: str) -> RawDocument:
        """
        MÉTODO 1: Carga un PDF desde disco (keyword-only params).
        ENTRADA: pdf_path (string) - keyword only
        PROCESO:
          - Leer bytes del PDF
          - Validar que es PDF válido
          - Extraer metadata básica (autor, fecha, páginas)
        SALIDA: RawDocument {bytes, metadata, num_pages}
        SYNC

        Args:
            pdf_path: Ruta al archivo PDF (keyword-only)

        Returns:
            RawDocument con información básica del PDF

        Raises:
            FileNotFoundError: Si el archivo no existe
            ValueError: Si el archivo no es un PDF válido
            TypeError: If pdf_path is not a string
        """
        # Runtime validation at ingress
        if not isinstance(pdf_path, str):
            raise TypeError(
                f"ERR_CONTRACT_MISMATCH[fn=load_pdf, param='pdf_path', "
                f"expected=str, got={type(pdf_path).__name__}, "
                f"producer=caller, consumer=DocumentLoader.load_pdf]"
            )
        file_path = pdf_path
        pdf_path = Path(pdf_path)

        if not pdf_path.exists():
            raise FileNotFoundError(f"Archivo no encontrado: {file_path}")

        if pdf_path.suffix.lower() != '.pdf':
            raise ValueError(f"El archivo debe ser PDF: {file_path}")

        self.logger.info(f"Cargando PDF: {pdf_path.name}")

        # Calcular hash del archivo
        file_hash = self._calculate_file_hash(pdf_path)

        # Obtener información básica del archivo
        file_stats = pdf_path.stat()

        # Extraer metadata con PyPDF2
        try:
            reader = PdfReader(str(pdf_path))
            num_pages = len(reader.pages)
            metadata = self.extract_metadata(reader)
            is_valid = self.validate_pdf_reader(reader)

        except Exception as e:
            self.logger.error(f"Error leyendo PDF con PyPDF2: {e}")
            raise ValueError(f"PDF corrupto o inválido: {file_path}") from e

        raw_doc = RawDocument(
            file_path=str(pdf_path.absolute()),
            file_name=pdf_path.name,
            num_pages=num_pages,
            file_size_bytes=file_stats.st_size,
            file_hash=file_hash,
            metadata=_to_frozen_mapping(metadata),
            is_valid=is_valid,
        )

        self.logger.info(f"✓ PDF cargado: {num_pages} páginas, {file_stats.st_size /
1024:.1f} KB")

        return raw_doc
```

```
@calibrated_method("saaaaaa.processing.document_ingestion.DocumentLoader.validate_pdf")
    def validate_pdf(self, *, raw_doc: RawDocument) -> bool:
        """
        MÉTODO 2: Valida que el PDF sea procesable.

        Verificaciones:
        - Número de páginas > 0
        - Tamaño de archivo razonable (< 500 MB)
        - No está encriptado

        Args:
            raw_doc: Documento crudo a validar

        Returns:
            True si es válido, False si no
        """
        if raw_doc.num_pages == 0:
            self.logger.error("PDF no tiene páginas")
            return False

        # Validar tamaño (500 MB máximo)
        max_size = 500 * 1024 * 1024  # 500 MB
        if raw_doc.file_size_bytes > max_size:
            self.logger.warning(f"PDF muy grande: {raw_doc.file_size_bytes /
(1024*1024):.1f} MB")
            return False

        # Verificar si está encriptado
        if raw_doc.metadata.get('encrypted', False):
            self.logger.error("PDF está encriptado")
            return False

        return True

    @calibrated_method("saaaaaa.processing.document_ingestion.DocumentLoader.validate_pdf_
reader")
    def validate_pdf_reader(self, reader: PdfReader) -> bool:
        """Valida un PdfReader de PyPDF2."""
        if reader.is_encrypted:
            return False
        return len(reader.pages) != 0

    @calibrated_method("saaaaaa.processing.document_ingestion.DocumentLoader.extract_metad
ata")
    def extract_metadata(self, reader: PdfReader) -> dict[str, Any]:
        """
        MÉTODO 3: Extrae metadata del PDF.

        Args:
            reader: PdfReader de PyPDF2

        Returns:
            Diccionario con metadata del PDF
        """
        metadata = {}

        try:
            pdf_metadata = reader.metadata

            if pdf_metadata:
                metadata = {
                    'author': pdf_metadata.get('/Author', 'Desconocido'),
                    'creator': pdf_metadata.get('/Creator', 'Desconocido'),
                    'producer': pdf_metadata.get('/Producer', 'Desconocido'),
                    'subject': pdf_metadata.get('/Subject', ''),
                    'title': pdf_metadata.get('/Title', ''),
                'creation_date': str(pdf_metadata.get('/CreationDate', '')),
                'modification_date': str(pdf_metadata.get('/ModDate', ''))
            }

            metadata['encrypted'] = reader.is_encrypted
            metadata['page_count'] = len(reader.pages)

        except Exception as e:
            self.logger.warning(f"Error extrayendo metadata: {e}")

        return metadata

    @calibrated_method("saaaaaa.processing.document_ingestion.DocumentLoader._calculate_fi
le_hash")
    def _calculate_file_hash(self, file_path: Path) -> str:
        """Calcula hash SHA-256 del archivo para trazabilidad."""
        # Delegate to factory for I/O operation
        from .factory import calculate_file_hash
        return calculate_file_hash(file_path)

# ============================================================================
# CLASE 2: TextExtractor
# ============================================================================

class TextExtractor:
    """
    Extrae texto de PDFs preservando estructura.
    Usa pdfplumber como método primario.
    """

    def __init__(self) -> None:
        self.logger = logger

    @calibrated_method("saaaaaa.processing.document_ingestion.TextExtractor.extract_full_t
ext")
    def extract_full_text(self, *, raw_doc: RawDocument) -> str:
        """
        MÉTODO 4: Extrae todo el texto del PDF (keyword-only params).

        ENTRADA: RawDocument (keyword only)
        PROCESO:
          - Extraer texto de todas las páginas
          - Preservar estructura (párrafos, secciones)
          - Identificar headers/footers
        SALIDA: string (texto completo)
        SYNC

        Args:
            raw_doc: Documento crudo cargado (keyword-only)

        Returns:
            Texto completo del documento
        """
        self.logger.info(f"Extrayendo texto completo de: {raw_doc.file_name}")

        # Delegate to factory for I/O operation
        from .factory import extract_pdf_text_all_pages

        try:
            text = extract_pdf_text_all_pages(raw_doc.file_path)
            self.logger.info(f"✓ Texto extraído: {len(text)} caracteres")
            return text

        except Exception as e:
            self.logger.error(f"Error abriendo PDF con pdfplumber: {e}")
            raise
```

```
@calibrated_method("saaaaaa.processing.document_ingestion.TextExtractor.extract_by_page")
    def extract_by_page(self, *, raw_doc: RawDocument, page: int) -> str:
        """
        MÉTODO 5: Extrae texto de una página específica.

        Args:
            raw_doc: Documento crudo
            page: Número de página (1-indexed)

        Returns:
            Texto de la página especificada
        """
        if page < 1 or page > raw_doc.num_pages:
            raise ValueError(f"Página {page} fuera de rango (1-{raw_doc.num_pages})")

        # Delegate to factory for I/O operation
        from .factory import extract_pdf_text_single_page

        try:
            text = extract_pdf_text_single_page(raw_doc.file_path, page, 
raw_doc.num_pages)
            return text

        except Exception as e:
            self.logger.error(f"Error extrayendo página {page}: {e}")
            return ""

    @calibrated_method("saaaaaa.processing.document_ingestion.TextExtractor.preserve_struc
ture")
    def preserve_structure(self, *, text: str) -> StructuredTextV1:
        """
        MÉTODO 6: Preserva estructura del documento.

        Detecta:
        - Secciones principales (títulos en mayúsculas)
        - Subsecciones (títulos numerados)
        - Límites de páginas

        Args:
            text: Texto completo del documento

        Returns:
            StructuredText con jerarquía preservada
        """
        sections: list[MutableMapping[str, Any]] = []
        page_boundaries: list[tuple[int, int]] = []

        lines = text.split('\n')
        current_position = 0
        current_section: MutableMapping[str, Any] | None = None

        for line in lines:
            line_stripped = line.strip()

            # Detectar marcador de página
            if line_stripped.startswith('--- Página'):
                if current_section:
                    sections.append(current_section)
                    current_section = None

                int(line_stripped.split()[2])
                page_boundaries.append((current_position, current_position + len(line)))

            # Detectar título de sección (mayúsculas, > 10 caracteres)
            elif line_stripped.isupper() and len(line_stripped) > 10:
                if current_section:
                    sections.append(current_section)
                current_section = {
                    'title': line_stripped,
                    'start_char': current_position,
                    'content': ''
                }

            # Agregar contenido a sección actual
            elif current_section is not None:
                current_section['content'] += line + '\n'

            current_position += len(line) + 1  # +1 por \n

        # Agregar última sección
        if current_section:
            sections.append(current_section)

        structured_sections = tuple(
            StructuredSection(
                title=str(section.get('title', '')),
                start_char=int(section.get('start_char', 0)),
                content=str(section.get('content', '')),
            )
            for section in sections
        )
        structured_page_boundaries = tuple((int(start), int(end)) for start, end in
page_boundaries)

        return StructuredTextV1(
            full_text=text,
            sections=structured_sections,
            page_boundaries=structured_page_boundaries,
        )

# ============================================================================
# CLASE 3: PreprocessingEngine
# ============================================================================

class PreprocessingEngine:
    """
    Motor de preprocesamiento unificado.
    Coordina la transformación de RawDocument → PreprocessedDocument.
    """

    def __init__(self) -> None:
        self.logger = logger

        # Inicializar procesadores de módulos existentes
        if PolicyTextProcessor:
            self.text_processor = PolicyTextProcessor()
        else:
            self.text_processor = None
            self.logger.warning("PolicyTextProcessor no disponible")

        if PDETMunicipalPlanAnalyzer:
            self.table_analyzer = PDETMunicipalPlanAnalyzer()
        else:
            self.table_analyzer = None
            self.logger.warning("PDETMunicipalPlanAnalyzer no disponible")

    @calibrated_method("saaaaaa.processing.document_ingestion.PreprocessingEngine.preproce
ss_document")
    def preprocess_document(self, *, raw_doc: RawDocument) -> PreprocessedDocument:
        """
        MÉTODO 7: Pipeline completo de preprocesamiento (keyword-only params).

        ENTRADA: RawDocument (keyword only)
        PROCESO INTERNO (SYNC pero con llamadas a métodos existentes):
          1. Extraer texto completo
          2. Normalizar encoding (usa PP.PolicyTextProcessor.normalize_unicode)
          3. Segmentar en oraciones (usa PP.PolicyTextProcessor.segment_into_sentences)
          4. Extraer tablas (usa FV.PDETMunicipalPlanAnalyzer.extract_tables)
          5. Limpiar y clasificar tablas
          6. Construir índices
          7. Detectar idioma

        SALIDA: PreprocessedDocument (inmutable, cacheable)
        SYNC

        Args:
            raw_doc: Documento crudo cargado

        Returns:
            Documento completamente preprocesado
        """
        self.logger.info(f"Iniciando preprocesamiento: {raw_doc.file_name}")

        # PASO 1: Extraer texto completo
        text_extractor = TextExtractor()
        full_text = text_extractor.extract_full_text(raw_doc)
        structured_text = text_extractor.preserve_structure(full_text)

        # PASO 2: Normalizar encoding
        normalized_text = self.normalize_encoding(full_text)

        # PASO 3: Segmentar en oraciones
        if self.text_processor:
            sentences_data = self.text_processor.segment_into_sentences(normalized_text)

            # Extraer lista de oraciones y metadata
            if isinstance(sentences_data, dict):
                sentences = sentences_data.get('sentences', [])
                sentence_metadata = sentences_data.get('metadata', [])
            else:
                sentences = sentences_data
                sentence_metadata = [{'index': i} for i in range(len(sentences))]
        else:
            # Fallback simple
            sentences = [s.strip() for s in normalized_text.split('.') if s.strip()]
            sentence_metadata = [{'index': i} for i in range(len(sentences))]

        self.logger.info(f"✓ Segmentado en {len(sentences)} oraciones")

        # PASO 4: Extraer tablas
        tables = []
        if self.table_analyzer:
            try:
                raw_tables = self.table_analyzer.extract_tables(raw_doc.file_path)

                # PASO 5: Limpiar y clasificar tablas
                if raw_tables:
                    cleaned_tables = [
                        self.table_analyzer._clean_dataframe(table)
                        for table in raw_tables
                    ]
                    tables = self.table_analyzer._classify_tables(cleaned_tables)

                self.logger.info(f"✓ Extraídas {len(tables)} tablas")

            except Exception as e:
                self.logger.warning(f"Error extrayendo tablas: {e}")

        sentences_tuple: tuple[str, ...] = tuple(str(sentence) for sentence in sentences)
        sentence_metadata_entries = _build_sentence_metadata_entries(sentence_metadata,
sentences_tuple)
        table_annotations = _coerce_table_annotations(tables)
        # PASO 6: Construir índices
        indexes = self._build_document_indexes(sentences_tuple, table_annotations)

        # PASO 7: Detectar idioma con runtime config
        language, language_detection_info = self.detect_language(text=normalized_text)

        metadata_dict: MutableMapping[str, Any] = {
            'num_sentences': len(sentences_tuple),
            'num_tables': len(table_annotations),
            'text_length': len(normalized_text),
            'index_terms': len(indexes.term_index),
            'source_path': raw_doc.file_path,
            'file_hash': raw_doc.file_hash,
            'file_name': raw_doc.file_name,
            'page_count': raw_doc.num_pages,
            'language_detection_info': {
                'tier': language_detection_info.tier.value,
                'detected_language': language_detection_info.detected_language,
                'reason': language_detection_info.reason,
            },
        }

        preprocessed_doc = PreprocessedDocument(
            document_id=raw_doc.file_name,
            full_text=normalized_text,
            sentences=sentences_tuple,
            language=language,
            structured_text=structured_text,
            sentence_metadata=sentence_metadata_entries,
            tables=table_annotations,
            indexes=indexes,
            metadata=_to_frozen_mapping(metadata_dict),
            ingested_at=datetime.utcnow(),
        )

        self.logger.info("✓ Preprocesamiento completado")

        return preprocessed_doc

    @calibrated_method("saaaaaa.processing.document_ingestion.PreprocessingEngine.normaliz
e_encoding")
    def normalize_encoding(self, *, text: str) -> str:
        """
        MÉTODO 8: Normaliza encoding del texto.

        Delega a PP.PolicyTextProcessor.normalize_unicode()

        Args:
            text: Texto a normalizar

        Returns:
            Texto normalizado
        """
        if self.text_processor:
            return self.text_processor.normalize_unicode(text)
        else:
            # Fallback: normalización básica
            import unicodedata
            return unicodedata.normalize('NFC', text)

<<<<<<< HEAD
    def detect_language(
        self,
        *,
        text: str,
        runtime_config: Optional[RuntimeConfig] = None,
    ) -> tuple[str, LanguageDetectionInfo]:
```

```
=======
    @calibrated_method("saaaaaa.processing.document_ingestion.PreprocessingEngine.detect_l
anguage")
    def detect_language(self, *, text: str) -> str:
>>>>>>> 5a00e83cc3f9f5ba388e245e9f2ae1d8107bec42
        """
        MÉTODO 9: Detecta el idioma del documento con runtime config integration.

        Args:
            text: Texto a analizar
            runtime_config: Optional runtime configuration (uses global if None)

        Returns:
            Tuple of (language_code, LanguageDetectionInfo manifest)
        """
        if runtime_config is None:
            runtime_config = get_runtime_config()

        if not text or len(text.strip()) < 20:
            # Not enough text to detect language
            lang_info = LanguageDetectionInfo(
                tier=LanguageTier.WARN_DEFAULT_ES,
                detected_language='es',
                reason='Insufficient text for detection (< 20 chars)'
            )
            return 'es', lang_info

        try:
            # Usar muestra del texto
            sample = text[:5000] if len(text) > 5000 else text
            detected_lang = detect(sample)

            self.logger.info(f"✓ Idioma detectado: {detected_lang}")

            # Successful detection
            lang_info = LanguageDetectionInfo(
                tier=LanguageTier.NORMAL,
                detected_language=detected_lang,
                reason=None
            )
            return detected_lang, lang_info

        except LangDetectException as e:
            # Category B fallback: Quality degradation but acceptable
            self.logger.warning("No se pudo detectar idioma, asumiendo español")

            lang_info = LanguageDetectionInfo(
                tier=LanguageTier.WARN_DEFAULT_ES,
                detected_language='es',
                reason=f'LangDetectException: {str(e)}'
            )

            # Emit structured log and metrics
            log_fallback(
                component='language_detection',
                subsystem='document_ingestion',
                fallback_category=FallbackCategory.B,
                fallback_mode='warn_default_es',
                reason=f'LangDetectException: {str(e)}',
                runtime_mode=runtime_config.mode,
            )

            increment_fallback(
                component='language_detection',
                fallback_category=FallbackCategory.B,
                fallback_mode='warn_default_es',
                runtime_mode=runtime_config.mode,
            )
            return 'es', lang_info

        except Exception as e:
            # Category B fallback: Unexpected error
            self.logger.error(f"Error detectando idioma: {e}")

            lang_info = LanguageDetectionInfo(
                tier=LanguageTier.FAIL,
                detected_language='unknown',
                reason=f'Unexpected error: {str(e)}'
            )

            # Emit structured log and metrics
            log_fallback(
                component='language_detection',
                subsystem='document_ingestion',
                fallback_category=FallbackCategory.B,
                fallback_mode='fail',
                reason=f'Unexpected error: {str(e)}',
                runtime_mode=runtime_config.mode,
            )

            increment_fallback(
                component='language_detection',
                fallback_category=FallbackCategory.B,
                fallback_mode='fail',
                runtime_mode=runtime_config.mode,
            )

            return 'unknown', lang_info

    def _build_document_indexes(
        self,
        sentences: Sequence[str],
        tables: Sequence[TableAnnotation],
    ) -> DocumentIndexesV1:
        """
        Construye índices sobre el documento para búsqueda rápida.

        INCLUYE:
        - Índice invertido de términos
        - Índice de números
        - Índice de marcadores temporales
        - Índice de entidades

        Args:
            sentences: Lista de oraciones
            tables: Lista de tablas clasificadas

        Returns:
            DocumentIndexes con todos los índices
        """
        term_index: MutableMapping[str, list[int]] = {}
        numeric_index: MutableMapping[str, list[int]] = {}
        temporal_index: MutableMapping[str, list[int]] = {}
        entity_index: MutableMapping[str, list[int]] = {}

        import re

        # Construir índices iterando sobre oraciones
        for sent_idx, sentence in enumerate(sentences):
            sentence_lower = sentence.lower()

            # Índice de términos (palabras > 3 caracteres)
            words = re.findall(r'\b\w{4,}\b', sentence_lower)
            for word in set(words):
                if word not in term_index:
                    term_index[word] = []
                term_index[word].append(sent_idx)

            # Índice de números
            numbers = re.findall(r'\d+(?:\.\d+)?', sentence)
            for num_str in numbers:
                numeric_index.setdefault(num_str, []).append(sent_idx)

            # Índice temporal (años, fechas)
            years = re.findall(r'\b(20\d{2})\b', sentence)
            for year in years:
                temporal_index.setdefault(year, []).append(sent_idx)

            # Índice de entidades (palabras capitalizadas)
            entities = re.findall(r'\b[A-ZÁÉÍÓÚÑ][a-záéíóúñ]+\b', sentence)
            for entity in set(entities):
                entity_index.setdefault(entity, []).append(sent_idx)

        for table in tables:
            entity_index.setdefault(table.label, []).append(-1)

        term_index_frozen = MappingProxyType({key: tuple(sorted(set(ids))) for key, ids in
 term_index.items()})
        numeric_index_frozen = MappingProxyType({key: tuple(sorted(set(ids))) for key, ids
 in numeric_index.items()})
        temporal_index_frozen = MappingProxyType({key: tuple(sorted(set(ids))) for key,
ids in temporal_index.items()})
        entity_index_frozen = MappingProxyType({key: tuple(sorted(set(ids))) for key, ids
in entity_index.items()})

        self.logger.info(f"✓ Índices construidos: {len(term_index)} términos,
{len(numeric_index)} números")

        return DocumentIndexesV1(
            term_index=term_index_frozen,
            numeric_index=numeric_index_frozen,
            temporal_index=temporal_index_frozen,
            entity_index=entity_index_frozen,
        )

# ===========================================================================
# FUNCIÓN DE CONVENIENCIA
# ===========================================================================

def ingest_document(*, pdf_path: str) -> PreprocessedDocument:
    """
    Función de conveniencia para ejecutar pipeline completo de ingesta.

    Args:
        pdf_path: Ruta al archivo PDF

    Returns:
        PreprocessedDocument listo para evaluación
    """
    # Paso 1: Cargar PDF
    loader = DocumentLoader()
    raw_doc = loader.load_pdf(pdf_path)

    # Paso 2: Validar
    if not loader.validate_pdf(raw_doc):
        raise ValueError(f"PDF no válido: {pdf_path}")

    # Paso 3: Preprocesar
    engine = PreprocessingEngine()
    preprocessed_doc = engine.preprocess_document(raw_doc)

    return preprocessed_doc
```

```
# ============================================================================
# EJEMPLO DE USO
# ============================================================================
# Example usage has been moved to examples/ directory to keep core modules pure

===== FILE: src/saaaaaa/processing/embedding_policy.py =====
"""
INTERNAL SPC COMPONENT

⚠  USAGE RESTRICTION ⚠
==============================================================================
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:
    scripts/smart_policy_chunks_canonic_phase_one.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:
    - smart_policy_chunks_canonic_phase_one.py
    - 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

import hashlib
import logging
import re
from dataclasses import dataclass
from enum import Enum
from functools import lru_cache
from typing import TYPE_CHECKING, Any, Literal, Protocol, TypedDict

import numpy as np
from sentence_transformers import CrossEncoder, SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from collections.abc import Iterable

    from numpy.typing import NDArray

# ============================================================================
# DESIGN CONSTANTS - Model Configuration
# ============================================================================

# Model constants
DEFAULT_CROSS_ENCODER_MODEL = "cross-encoder/ms-marco-MiniLM-L-6-v2"
MODEL_PARAPHRASE_MULTILINGUAL = "sentence-transformers/paraphrase-multilingual-mpnet-base-v2"

# ============================================================================
# TYPE SYSTEM - Python 3.10+ Type Safety
# ============================================================================

class PolicyDomain(Enum):
    """
    Colombian PDM policy areas (PA01-PA10) per canonical notation.

    Values are loaded from questionnaire_monolith.json canonical_notation.
    Use CanonicalPolicyArea from saaaaaa.core.canonical_notation for dynamic access.
    """

    # Legacy IDs mapped to canonical codes for backward compatibility
    P1 = "PA01"  # Derechos de las mujeres e igualdad de género
    P2 = "PA02"  # Prevención de la violencia y protección frente al conflicto
    P3 = "PA03"  # Ambiente sano, cambio climático, prevención y atención a desastres
    P4 = "PA04"  # Derechos económicos, sociales y culturales
    P5 = "PA05"  # Derechos de las víctimas y construcción de paz
    P6 = "PA06"  # Derecho al buen futuro de la niñez, adolescencia, juventud
    P7 = "PA07"  # Tierras y territorios
    P8 = "PA08"  # Líderes y defensores de derechos humanos
    P9 = "PA09"  # Crisis de derechos de personas privadas de la libertad
    P10 = "PA10"  # Migración transfronteriza

class AnalyticalDimension(Enum):
    """
    Analytical dimensions (D1-D6) per canonical notation.

    Values reference canonical notation from questionnaire_monolith.json.
    Use CanonicalDimension from saaaaaa.core.canonical_notation for dynamic access.
    """

    D1 = "DIM01"  # INSUMOS - Diagnóstico y Recursos
    D2 = "DIM02"  # ACTIVIDADES - Diseño de Intervención
    D3 = "DIM03"  # PRODUCTOS - Productos y Outputs
    D4 = "DIM04"  # RESULTADOS - Resultados y Outcomes
    D5 = "DIM05"  # IMPACTOS - Impactos de Largo Plazo
    D6 = "DIM06"  # CAUSALIDAD - Teoría de Cambio

class PDQIdentifier(TypedDict):
    """Canonical P-D-Q identifier structure."""

    question_unique_id: str  # P#-D#-Q#
    policy: str  # P#
    dimension: str  # D#
    question: int  # Q#
    rubric_key: str  # D#-Q#

class PosteriorSampleRecord(TypedDict):
    """Serializable posterior sample used by downstream Bayesian consumers."""

    coherence: float

class SemanticChunk(TypedDict):
    """Structured semantic chunk with metadata."""

    chunk_id: str
    content: str
    embedding: NDArray[np.float32]
    metadata: dict[str, Any]
    pdq_context: PDQIdentifier | None
    token_count: int
    position: tuple[int, int]  # (start, end) in document

class PosteriorSample(TypedDict):
    """Serialized posterior sample representation."""

    coherence: float

class BayesianEvaluation(TypedDict):
    """Bayesian uncertainty-aware evaluation result."""

    point_estimate: float  # 0.0-1.0
    credible_interval_95: tuple[float, float]
    posterior_samples: list[PosteriorSample]
    evidence_strength: Literal["weak", "moderate", "strong", "very_strong"]
    numerical_coherence: float  # Statistical consistency score
    posterior_records: list[PosteriorSampleRecord]

class EmbeddingProtocol(Protocol):
    """Protocol for embedding models."""

    def encode(
        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"^(?:CAPÍTULO|SECCIÓN|ARTÍCULO|PROGRAMA|PROYECTO|EJE)\s+[IVX\d]+",
    re.MULTILINE | re.IGNORECASE,
)
TABLE_MARKERS = re.compile(r"(?:Tabla|Cuadro|Figura)\s+\d+", re.IGNORECASE)
LIST_MARKERS = re.compile(r"^[\s]*[•\-\*\d]+[\.\)]\s+", re.MULTILINE)
NUMERIC_INDICATORS = re.compile(
    r"\b\d+(?:[.,]\d+)?(?:\s*%|millones?|mil|billones?)?\b", re.IGNORECASE
)

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

def chunk_document(
    self,
    *,
    text: str,
    document_metadata: dict[str, Any],
) -> list[SemanticChunk]:
    """
    Chunk document with advanced semantic awareness (keyword-only params).

    Args:
        text: Document text to chunk
        document_metadata: Metadata dict with at least 'doc_id' key

    Returns:
        List of semantic chunks with preserved structure and P-D-Q context

    Raises:
        TypeError: If text is not a string
        KeyError: If document_metadata missing required keys
    """
    # Runtime validation at ingress
    if not isinstance(text, str):
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH[fn=chunk_document, param='text', "
            f"expected=str, got={type(text).__name__}]"
        )

    if not isinstance(document_metadata, dict):
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH[fn=chunk_document, param='document_metadata', "
            f"expected=dict, got={type(document_metadata).__name__}]"
        )
    # Preprocess: normalize whitespace, preserve structure
    normalized_text = self._normalize_text(text)

    # Extract structural elements
    sections = self._extract_sections(normalized_text)
    tables = self._extract_tables(normalized_text)
    lists = self._extract_lists(normalized_text)

    # Generate chunks with boundary preservation
    raw_chunks = self._recursive_split(
        normalized_text,
        target_size=self.config.chunk_size,
        overlap=self.config.chunk_overlap,
    )

    # Enrich chunks with metadata and P-D-Q context
    semantic_chunks: list[SemanticChunk] = []

    for idx, chunk_text in enumerate(raw_chunks):
        # Infer P-D-Q context from chunk text
        pdq_context = self._infer_pdq_context(chunk_text)
```

```
            # Count tokens (approximation: Spanish has ~1.3 chars/token)
            AVG_CHARS_PER_TOKEN = 1.3  # Source: Spanish language statistics
            token_count = int(
                len(chunk_text) / AVG_CHARS_PER_TOKEN
            )  # Approximate token count

            # Create structured chunk
            chunk_id = hashlib.sha256(
                f"{document_metadata.get('doc_id', '')}_{idx}_{chunk_text[:50]}".encode()
            ).hexdigest()[:16]

            semantic_chunk: SemanticChunk = {
                "chunk_id": chunk_id,
                "content": chunk_text,
                "embedding": np.array([]),  # Filled later
                "metadata": {
                    "document_id": document_metadata.get("doc_id"),
                    "chunk_index": idx,
                    "has_table": self._contains_table(chunk_text, tables),
                    "has_list": self._contains_list(chunk_text, lists),
                    "has_numbers": bool(self.NUMERIC_INDICATORS.search(chunk_text)),
                    "section_title": self._find_section(chunk_text, sections),
                },
                "pdq_context": pdq_context,
                "token_count": token_count,
                "position": (0, len(chunk_text)),  # Updated during splitting
            }

            semantic_chunks.append(ensure_content_schema(semantic_chunk))

        self._logger.info(
            "Created %d semantic chunks from document %s",
            len(semantic_chunks),
            document_metadata.get("doc_id", "unknown"),
        )

        return semantic_chunks

    @calibrated_method("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._norma
lize_text")
    def _normalize_text(self, text: str) -> str:
        """Normalize text while preserving structure."""
        # Remove excessive whitespace but preserve paragraph breaks
        text = re.sub(r"[ \t]+", " ", text)
        text = re.sub(r"\n{3,}", "\n\n", text)
        return text.strip()

    @calibrated_method("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._recur
sive_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("saaaaaa.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("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._extra
ct_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("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._extra
ct_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("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._extra
ct_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", "género", "igualdad", "equidad"],
            "PA02": ["violencia", "conflicto", "seguridad", "prevención"],
            "PA03": ["ambiente", "clima", "desastre", "riesgo"],
            "PA04": ["económico", "social", "cultural", "empleo"],
            "PA05": ["víctima", "paz", "reconciliación", "reparación"],
            "PA06": ["niñez", "adolescente", "juventud", "futuro"],
            "PA07": ["tierra", "territorio", "rural", "agrario"],
            "PA08": ["líder", "defensor", "derechos humanos"],
            "PA09": ["privado libertad", "cárcel", "reclusión"],
            "PA10": ["migración", "frontera", "venezolano"],
        }

        dimension_keywords = {
            "DIM01": ["diagnóstico", "baseline", "situación", "recurso"],
            "DIM02": ["diseño", "estrategia", "intervención", "actividad"],
            "DIM03": ["producto", "output", "entregable", "meta"],
            "DIM04": ["resultado", "outcome", "efecto", "cambio"],
            "DIM05": ["impacto", "largo plazo", "sostenibilidad"],
            "DIM06": ["teoría", "causal", "coherencia", "lógica"],
        }

        # Score policies and dimensions
        policy_scores = {
            policy: sum(1 for kw in keywords if kw.lower() in chunk_text.lower())
            for policy, keywords in policy_keywords.items()
        }

        dimension_scores = {
            dim: sum(1 for kw in keywords if kw.lower() in chunk_text.lower())
            for dim, keywords in dimension_keywords.items()
        }

        # Select best match if confidence is sufficient
        best_policy = max(policy_scores, key=policy_scores.get)
        best_dimension = max(dimension_scores, key=dimension_scores.get)

        if policy_scores[best_policy] > 0 and dimension_scores[best_dimension] > 0:
            # Generate canonical identifier
            question_num = 1  # Simplified; real system would infer from context
            question_code = f"Q{question_num:03d}"

            return PDQIdentifier(
                question_unique_id=f"{best_policy}-{best_dimension}-{question_code}",
                policy=best_policy,
                dimension=best_dimension,
                question=question_num,
                rubric_key=f"{best_dimension}-{question_code}",
            )

        return None

    def _contains_table(
        self, chunk_text: str, tables: list[dict[str, Any]]
    ) -> bool:
        """Check if chunk contains table markers."""
        return any(
            table["marker"] in chunk_text
            for table in tables
        )

    @calibrated_method("saaaaaa.processing.embedding_policy.AdvancedSemanticChunker._conta
ins_list")
    def _contains_list(self, chunk_text: str, lists: list[dict[str, Any]]) -> bool:
        """Check if chunk contains list structures."""
        return bool(self.LIST_MARKERS.search(chunk_text))

    def _find_section(
        self, chunk_text: str, sections: list[dict[str, Any]]
    ) -> str | None:
        """Find section title for chunk."""
        # Simplified: would use position-based matching in production
        for section in sections:
            if section["title"][:20] in chunk_text:
                return section["title"]
        return None

# ============================================================================
# BAYESIAN NUMERICAL ANALYSIS - Rigorous Statistical Framework
# ============================================================================

class BayesianNumericalAnalyzer:
    """
    Bayesian framework for uncertainty-aware numerical policy analysis.

    Implements:
    - Beta-Binomial conjugate prior for proportions
    - Normal-Normal conjugate prior for continuous metrics
    - Bayesian hypothesis testing for policy comparisons
    - Credible interval estimation
    - Evidence strength quantification (Bayes factors)
    """

    def __init__(self, prior_strength: float = 1.0) -> None:
        """
        Initialize Bayesian analyzer.

        Args:
            prior_strength: Prior belief strength (get_parameter_loader().get("saaaaaa.pro
cessing.embedding_policy.BayesianNumericalAnalyzer.__init__").get("auto_param_L510_51",
1.0) = weak, 1get_parameter_loader().get("saaaaaa.processing.embedding_policy.BayesianNume
ricalAnalyzer.__init__").get("auto_param_L510_64", 0.0) = strong)
        """
        self.prior_strength = prior_strength
        self._logger = logging.getLogger(self.__class__.__name__)
        self._rng = np.random.default_rng()

    def evaluate_policy_metric(
        self,
        observed_values: list[float],
        n_posterior_samples: int = 10000,
        **kwargs: Any
    ) -> BayesianEvaluation:
        """
        Bayesian evaluation of policy metric with uncertainty quantification.
        Returns posterior distribution, credible intervals, and evidence strength.

        Args:
            observed_values: List of observed metric values
            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(α, β)
        Likelihood: Binomial
        Posterior: Beta(α + successes, β + failures)
        """
        # Prior parameters (weakly informative)
        alpha_prior = self.prior_strength
        beta_prior = self.prior_strength

        # Convert proportions to successes/failures
        n_obs = len(observations)
        sum_success = np.sum(observations)  # If already in [0,1]

        # Posterior parameters
        alpha_post = alpha_prior + sum_success
        beta_post = beta_prior + (n_obs - sum_success)

        # Sample from posterior
        posterior_samples = self._rng.beta(alpha_post, beta_post, size=n_samples)

        return posterior_samples.astype(np.float32)

    def _normal_normal_posterior(
        self, observations: NDArray[np.float32], n_samples: int
    ) -> NDArray[np.float32]:
        """
        Normal-Normal conjugate posterior for continuous metrics.

        Prior: Normal(μ₀, σ₀²)
        Likelihood: Normal(μ, σ²)
        Posterior: Normal(μ_post, σ_post²)
        """
        n_obs = len(observations)
        obs_mean = np.mean(observations)
        obs_std = np.std(observations, ddof=1) if n_obs > 1 else get_parameter_loader().ge
t("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer.__init__").get("auto_para
m_L616_65", 1.0)

        # Prior parameters (weakly informative centered on observed mean)
        mu_prior = obs_mean
        sigma_prior = obs_std * self.prior_strength

        # Posterior parameters (conjugate update)
        precision_prior = 1 / (sigma_prior**2)
        precision_likelihood = n_obs / (obs_std**2)

        precision_post = precision_prior + precision_likelihood
        mu_post = (
            precision_prior * mu_prior + precision_likelihood * obs_mean
        ) / precision_post
        sigma_post = np.sqrt(1 / precision_post)

        # Sample from posterior
        posterior_samples = self._rng.normal(mu_post, sigma_post, size=n_samples)

        return posterior_samples.astype(np.float32)

    def _classify_evidence_strength(
        self, credible_interval_width: float, **kwargs: Any
    ) -> Literal["weak", "moderate", "strong", "very_strong"]:
        """Classify evidence strength based on posterior uncertainty.

        Args:
            credible_interval_width: Width of the 95% credible interval
            **kwargs: Additional optional parameters for compatibility

        Returns:
            Evidence strength classification (weak/moderate/strong/very_strong)
        """
        if credible_interval_width > get_parameter_loader().get("saaaaaa.processing.embedd
ing_policy.BayesianNumericalAnalyzer.__init__").get("auto_param_L649_37", 0.5):
            return "weak"
        elif credible_interval_width > get_parameter_loader().get("saaaaaa.processing.embe
dding_policy.BayesianNumericalAnalyzer.__init__").get("auto_param_L651_39", 0.3):
            return "moderate"
        elif credible_interval_width > get_parameter_loader().get("saaaaaa.processing.embe
dding_policy.BayesianNumericalAnalyzer.__init__").get("auto_param_L653_39", 0.15):
            return "strong"
        else:
            return "very_strong"

    @calibrated_method("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer._com
pute_coherence")
    def _compute_coherence(self, observations: NDArray[np.float32], **kwargs: Any) ->
float:
        """
        Compute numerical coherence (consistency) score.

        Uses coefficient of variation and statistical tests.

        Args:
            observations: Array of observed values
            **kwargs: Additional optional parameters for compatibility

        Returns:
            Coherence score in [0, 1]
        """
        if len(observations) < 2:
            return get_parameter_loader().get("saaaaaa.processing.embedding_policy.Bayesia
nNumericalAnalyzer._compute_coherence").get("auto_param_L673_19", 1.0)

        # Coefficient of variation
        mean_val = np.mean(observations)
        std_val = np.std(observations, ddof=1)

        if mean_val == 0:
            return get_parameter_loader().get("saaaaaa.processing.embedding_policy.Bayesia
nNumericalAnalyzer._compute_coherence").get("auto_param_L680_19", 0.0)

        cv = std_val / abs(mean_val)

        # Normalize: lower CV = higher coherence
        coherence = np.exp(-cv)  # Exponential decay

        return float(np.clip(coherence, get_parameter_loader().get("saaaaaa.processing.emb
edding_policy.BayesianNumericalAnalyzer._compute_coherence").get("auto_param_L687_40",
0.0), get_parameter_loader().get("saaaaaa.processing.embedding_policy.BayesianNumericalAna
lyzer._compute_coherence").get("auto_param_L687_45", 1.0)))

    @calibrated_method("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer._nul
l_evaluation")
    def _null_evaluation(self) -> BayesianEvaluation:
        """Return null evaluation when no data available."""
        null_samples = to_dict_samples(np.array([get_parameter_loader().get("saaaaaa.proce
ssing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L692_4
9", 0.0)], dtype=np.float32))

        return BayesianEvaluation(
            point_estimate=get_parameter_loader().get("saaaaaa.processing.embedding_policy
.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L695_27", 0.0),
            credible_interval_95=(get_parameter_loader().get("saaaaaa.processing.embedding
_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L696_34", 0.0), get_p
arameter_loader().get("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer._null
_evaluation").get("auto_param_L696_39", 0.0)),
            posterior_samples=null_samples,
            evidence_strength="weak",
            numerical_coherence=get_parameter_loader().get("saaaaaa.processing.embedding_p
olicy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L699_32", 0.0),
            posterior_records=[{"coherence": get_parameter_loader().get("saaaaaa.processin
g.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L700_45",
0.0)}],
        )

    def serialize_posterior_samples(
        self, samples: NDArray[np.float32]
    ) -> list[PosteriorSampleRecord]:
        """Convert posterior samples into standardized coherence records.

        Safely handles None or non-array inputs and limits the number of
        serialized records to avoid excessive memory use.
        """
        if samples is None:
            return []

        # Ensure a 1-D numpy array of floats
        arr = np.asarray(samples, dtype=np.float32).ravel()

        # Prevent accidental excessive memory use when serializing huge arrays
        MAX_RECORDS = 10000
        values = arr.tolist()
        if len(values) > MAX_RECORDS:
            values = values[:MAX_RECORDS]

        return [{"coherence": float(v)} for v in values]

    def compare_policies(
        self,
        policy_a_values: list[float],
        policy_b_values: list[float],
    ) -> dict[str, Any]:
        """
        Bayesian comparison of two policy metrics.

        Returns probability that A > B and Bayes factor.
        """
        if not policy_a_values or not policy_b_values:
            return {"probability_a_better": get_parameter_loader().get("saaaaaa.processing
.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L736_44",
0.5), "bayes_factor": get_parameter_loader().get("saaaaaa.processing.embedding_policy.Baye
sianNumericalAnalyzer._null_evaluation").get("auto_param_L736_65", 1.0)}

        # Get posterior distributions
        eval_a = self.evaluate_policy_metric(policy_a_values)
        eval_b = self.evaluate_policy_metric(policy_b_values)

        # Compute probability that A > B and clip to avoid exact 0/1 which can cause
        # division-by-zero in subsequent Bayes factor calculation
        samples_a = samples_to_array(eval_a["posterior_samples"])
        samples_b = samples_to_array(eval_b["posterior_samples"])

        # Compute probability that A > B and clip to avoid exact 0/1 which can cause
        # division-by-zero in subsequent Bayes factor calculation.
        prob_a_better = float(np.mean(samples_a > samples_b))
        prob_a_better = float(np.clip(prob_a_better, 1e-6, get_parameter_loader().get("saa
aaaa.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_pa
ram_L750_59", 1.0) - 1e-6))

        # Compute Bayes factor (simplified)
        if prob_a_better > get_parameter_loader().get("saaaaaa.processing.embedding_policy
.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L753_27", 0.5):
            bayes_factor = prob_a_better / (1 - prob_a_better)
        else:
            bayes_factor = (1 - prob_a_better) / prob_a_better

        return {
            "probability_a_better": float(prob_a_better),
            "bayes_factor": float(bayes_factor),
            "difference_mean": float(np.mean(samples_a - samples_b)),
            "difference_ci_95": (
                float(
                    np.percentile(
                        samples_a - samples_b,
                        2.5,
                    )
                ),
                float(
                    np.percentile(
                        samples_a - samples_b,
                        97.5,
                    )
                ),
            ),
        }

# ================================================================================
# CROSS-ENCODER RERANKING - State-of-the-Art Retrieval
# ================================================================================

class PolicyCrossEncoderReranker:
    """
    Cross-encoder reranking optimized for Spanish policy documents.

    Uses transformer-based cross-attention for precise relevance scoring.
    Superior to bi-encoder + cosine similarity for final ranking.
    """

    def __init__(
        self,
        model_name: str = DEFAULT_CROSS_ENCODER_MODEL,
        max_length: int = 512,
        retry_handler=None,
    ) -> None:
        """
        Initialize cross-encoder reranker.

        Args:
            model_name: HuggingFace model name (multilingual preferred)
            max_length: Maximum sequence length for cross-encoder
            retry_handler: Optional RetryHandler for model loading

        Raises:
            RuntimeError: If online model download is required but HF_ONLINE=0
        """
        self._logger = logging.getLogger(self.__class__.__name__)
        self.retry_handler = retry_handler

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

        # Check if we're trying to download a remote model when offline
        if not _is_model_cached(model_name):
            lockdown.check_online_model_access(
                model_name=model_name,
                operation="load CrossEncoder model"
            )

        # Load model with retry logic if available
        if retry_handler:
            try:
                from retry_handler import DependencyType

                @retry_handler.with_retry(
                    DependencyType.EMBEDDING_SERVICE,
                    operation_name="load_cross_encoder",
                    exceptions=(OSError, IOError, ConnectionError, RuntimeError)
                )
                def load_model():
                    return CrossEncoder(model_name, max_length=max_length)

                self.model = load_model()
                self._logger.info(f"Cross-encoder loaded with retry protection:
{model_name}")
            except Exception as e:
                self._logger.error(f"Failed to load cross-encoder: {e}")
                raise
        else:
            self.model = CrossEncoder(model_name, max_length=max_length)
            self._logger.info(f"Cross-encoder loaded: {model_name}")

    def rerank(
        self,
        query: str,
        candidates: list[SemanticChunk],
        top_k: int = 10,
        min_score: float = get_parameter_loader().get("saaaaaa.processing.embedding_policy
.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L848_27", 0.0),
    ) -> list[tuple[SemanticChunk, float]]:
        """
        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 = get_parameter_loader().get("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("auto_param_L900_24", 0.7)  # Diversity vs relevance trade-off

    # Bayesian analysis
    prior_strength: float = get_parameter_loader().get("saaaaaa.processing.embedding_policy.BayesianNumericalAnalyzer._null_evaluation").get("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 saaaaaa.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)
        self._logger.info(
            "Filtered to %d chunks matching P-D-Q context", len(candidates)
        )

    if not candidates:
        return []

    # Cross-encoder reranking for precision
    if use_reranking:
        reranked = self.cross_encoder.rerank(
            query, candidates, top_k=self.config.top_k_rerank
        )
    else:
        # Use bi-encoder scores
        candidate_indices = [document_chunks.index(c) for c in candidates]
        reranked = [
            (candidates[i], float(similarities[candidate_indices[i]]))
            for i in range(len(candidates))
        ]
        reranked.sort(key=lambda x: x[1], reverse=True)
        reranked = reranked[: self.config.top_k_rerank]

    # MMR diversification
    if len(reranked) > 1:
        reranked = self._apply_mmr(reranked)
        return reranked

    def evaluate_policy_numerical_consistency(
        self,
        chunks: list[SemanticChunk],
        pdq_context: PDQIdentifier,
    ) -> BayesianEvaluation:
        """
        Bayesian evaluation of numerical consistency for policy metric.

        Extracts numerical values from chunks matching P-D-Q context,
        performs rigorous statistical analysis with uncertainty quantification.

        Args:
            chunks: Document chunks to analyze
            pdq_context: P-D-Q context to filter relevant chunks

        Returns:
            Bayesian evaluation with credible intervals and evidence strength
        """
        # Filter chunks by P-D-Q context
        relevant_chunks = self._filter_by_pdq(chunks, pdq_context)

        if not relevant_chunks:
            self._logger.warning(
                "No chunks found for P-D-Q context: %s",
                pdq_context["question_unique_id"],
            )
            return self.bayesian_analyzer._null_evaluation()

        # Extract numerical values from chunks
        numerical_values = self._extract_numerical_values(relevant_chunks)

        if not numerical_values:
            self._logger.warning(
                "No numerical values extracted from %d chunks", len(relevant_chunks)
            )
            return self.bayesian_analyzer._null_evaluation()

        # Perform Bayesian evaluation
        evaluation = self.bayesian_analyzer.evaluate_policy_metric(numerical_values)

        self._logger.info(
            "Evaluated %d numerical values for %s: point_estimate=%.3f, CI=[%.3f, %.3f],
evidence=%s",
            len(numerical_values),
            pdq_context["rubric_key"],
            evaluation["point_estimate"],
            evaluation["credible_interval_95"][0],
            evaluation["credible_interval_95"][1],
            evaluation["evidence_strength"],
        )

        return evaluation

    def compare_policy_interventions(
        self,
        intervention_a_chunks: list[SemanticChunk],
        intervention_b_chunks: list[SemanticChunk],
        pdq_context: PDQIdentifier,
    ) -> dict[str, Any]:
        """
        Bayesian comparison of two policy interventions.

        Returns probability and evidence for superiority.
        """
        values_a = self._extract_numerical_values(
            self._filter_by_pdq(intervention_a_chunks, pdq_context)
        )
        values_b = self._extract_numerical_values(
            self._filter_by_pdq(intervention_b_chunks, pdq_context)
        )

        return self.bayesian_analyzer.compare_policies(values_a, values_b)

    def generate_pdq_report(
        self,
        document_chunks: list[SemanticChunk],
        target_pdq: PDQIdentifier,
    ) -> dict[str, Any]:
        """
        Generate comprehensive analytical report for P-D-Q question.

        Combines semantic search, numerical analysis, and evidence synthesis.
        """
        # Semantic search for relevant content
        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("saaaaaa.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)
            return type(value).__name__
        def _log_mismatch(key: str, needed: Any, got: Any, index: int | None = None) ->
None:
            message = (
                "ERR_CONTRACT_MISMATCH[fn=_filter_by_pdq, "
                f"key='{key}', needed={_repr_contract(needed)}, got={_repr_contract(got)}"
            )
            if index is not None:
                message += f", index={index}"
            message += "]"
            self._logger.error(message)

        self._logger.debug(
            "edge %s → _filter_by_pdq | params=%s",
            self.__class__.__name__,
            {
                "chunks_type": type(chunks).__name__,
                "chunks_len": len(chunks) if isinstance(chunks, list) else "n/a",
                "pdq_filter_type": type(pdq_filter).__name__,
                "pdq_filter_keys": sorted(pdq_filter.keys())
                if isinstance(pdq_filter, dict)
                else None,
            },
        )

        if not isinstance(chunks, list):
            _log_mismatch("chunks", "list", chunks)
            return []

        if not isinstance(pdq_filter, dict):
            _log_mismatch("pdq_filter", "dict", pdq_filter)
            return []

        expected_policy = pdq_filter.get("policy")
        expected_dimension = pdq_filter.get("dimension")

        if expected_policy is None or expected_dimension is None:
            _log_mismatch(
                "pdq_filter",
                "keys=('policy','dimension')",
                {"policy": expected_policy, "dimension": expected_dimension},
            )
            return []

        filtered_chunks: list[SemanticChunk] = []

        for index, chunk in enumerate(chunks):
            if not isinstance(chunk, dict):
                _log_mismatch("chunk", "dict", chunk, index)
                continue

            pdq_context = chunk.get("pdq_context")

            if not pdq_context:
                _log_mismatch("pdq_context", True, pdq_context, index)
                continue

            if not isinstance(pdq_context, dict):
                _log_mismatch("pdq_context", "dict", pdq_context, index)
                continue

            policy = pdq_context.get("policy")
            dimension = pdq_context.get("dimension")

            if policy is None or dimension is None:
                _log_mismatch(
                    "pdq_context",
                    "keys=('policy','dimension')",
                    {"policy": policy, "dimension": dimension},
                    index,
                )
                continue

            if policy == expected_policy and dimension == expected_dimension:
                filtered_chunks.append(chunk)

        return filtered_chunks

    def _apply_mmr(
        self,
        ranked_results: list[tuple[SemanticChunk, float]],
    ) -> list[tuple[SemanticChunk, float]]:
        """
        Apply Maximal Marginal Relevance for diversification.

        Balances relevance with diversity to avoid redundant results.
        """
        if len(ranked_results) <= 1:
            return ranked_results

        chunks, scores = zip(*ranked_results, strict=False)
        chunk_embeddings = np.vstack([c["embedding"] for c in chunks])

        selected_indices = []
        remaining_indices = list(range(len(chunks)))

        # Select first (most relevant)
        selected_indices.append(0)
        remaining_indices.remove(0)

        # Iteratively select diverse documents
        while remaining_indices and len(selected_indices) < len(chunks):
            best_mmr_score = float("-inf")
            best_idx = None

            for idx in remaining_indices:
                # Relevance score
                relevance = scores[idx]

                # Diversity: max similarity to selected
                similarities_to_selected = cosine_similarity(
                    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("saaaaaa.processing.embedding_policy.PolicyAnalysisEmbedder._extrac
t_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 / 10get_parameter_loader().get("saaaaaa.processi
ng.embedding_policy.PolicyAnalysisEmbedder._extract_numerical_values").get("auto_param_L14
74_46", 0.0)

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

                    except (ValueError, IndexError):
                        continue

        return numerical_values

    @calibrated_method("saaaaaa.processing.embedding_policy.PolicyAnalysisEmbedder._genera
te_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 get_parameter_loader().get("saaaaaa.processing.embedding_policy.PolicyA
nalysisEmbedder._generate_query_from_pdq").get("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 get_parameter_loader()
.get("saaaaaa.processing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq"
).get("auto_param_L1514_68", 0.0)
        )

        # Numerical confidence: based on evidence strength and coherence
        evidence_strength_map = {
            "weak": get_parameter_loader().get("saaaaaa.processing.embedding_policy.Policy
AnalysisEmbedder._generate_query_from_pdq").get("auto_param_L1519_20", 0.25),
            "moderate": get_parameter_loader().get("saaaaaa.processing.embedding_policy.Po
licyAnalysisEmbedder._generate_query_from_pdq").get("auto_param_L1520_24", 0.5),
            "strong": get_parameter_loader().get("saaaaaa.processing.embedding_policy.Poli
cyAnalysisEmbedder._generate_query_from_pdq").get("auto_param_L1521_22", 0.75),
            "very_strong": get_parameter_loader().get("saaaaaa.processing.embedding_policy
.PolicyAnalysisEmbedder._generate_query_from_pdq").get("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 = get_parameter_loader().get("saaaaaa.processing.embedding_poli
cy.PolicyAnalysisEmbedder._generate_query_from_pdq").get("auto_param_L1530_29", 0.6) *
semantic_confidence + get_parameter_loader().get("saaaaaa.processing.embedding_policy.Poli
cyAnalysisEmbedder._generate_query_from_pdq").get("auto_param_L1530_57", 0.4) *
numerical_confidence

        return float(np.clip(overall_confidence, get_parameter_loader().get("saaaaaa.proce
ssing.embedding_policy.PolicyAnalysisEmbedder._generate_query_from_pdq").get("auto_param_L
1532_49", 0.0), get_parameter_loader().get("saaaaaa.processing.embedding_policy.PolicyAnal
ysisEmbedder._generate_query_from_pdq").get("auto_param_L1532_54", 1.0)))

    @lru_cache(maxsize=1024)
    @calibrated_method("saaaaaa.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("saaaaaa.processing.embedding_policy.PolicyAnalysisEmbedder.get_dia
gnostics")
    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
# ============================================================================

def create_policy_embedder(
    model_tier: Literal["fast", "balanced", "accurate"] = "balanced",
) -> PolicyAnalysisEmbedder:
    """
    Factory function for creating production-ready policy embedder.

    Args:
        model_tier: Performance/accuracy trade-off
            - "fast": Lightweight, low latency
            - "balanced": Good performance/accuracy balance (default)
            - "accurate": Maximum accuracy, higher latency

    Returns:
        Configured PolicyAnalysisEmbedder instance
    """
    model_configs = {
        "fast": PolicyEmbeddingConfig(
            embedding_model="sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2",
            cross_encoder_model=DEFAULT_CROSS_ENCODER_MODEL,
            chunk_size=256,
            chunk_overlap=64,
            top_k_candidates=30,
            top_k_rerank=5,
            batch_size=64,
        ),
        "balanced": PolicyEmbeddingConfig(
            embedding_model=MODEL_PARAPHRASE_MULTILINGUAL,
            cross_encoder_model=DEFAULT_CROSS_ENCODER_MODEL,
            chunk_size=512,
            chunk_overlap=128,
            top_k_candidates=50,
            top_k_rerank=10,
            batch_size=32,
        ),
        "accurate": PolicyEmbeddingConfig(
            embedding_model=MODEL_PARAPHRASE_MULTILINGUAL,
            cross_encoder_model="cross-encoder/mmarco-mMiniLMv2-L12-H384-v1",
            chunk_size=768,
            chunk_overlap=192,
            top_k_candidates=100,
            top_k_rerank=20,
            batch_size=16,
        ),
    }

    config = model_configs[model_tier]

    logger = logging.getLogger("PolicyEmbedderFactory")
    logger.info("Creating policy embedder with tier: %s", model_tier)

    return PolicyAnalysisEmbedder(config)
```

```
# ============================================================================
# PRODUCER CLASS - Registry Exposure
# ============================================================================

class EmbeddingPolicyProducer:
    """
    Producer wrapper for embedding policy analysis with registry exposure

    Provides public API methods for orchestrator integration without exposing
    internal implementation details or summarization logic.

    Version: get_parameter_loader().get("saaaaaa.processing.embedding_policy.PolicyAnalysi
sEmbedder.get_diagnostics").get("auto_param_L1630_13", 1.0).0
    Producer Type: Embedding / Semantic Search
    """

    def __init__(
        self,
        config: PolicyEmbeddingConfig | None = None,
        model_tier: Literal["fast", "balanced", "accurate"] = "balanced",
        retry_handler=None
    ) -> None:
        """Initialize producer with optional configuration"""
        if config is None:
            self.embedder = create_policy_embedder(model_tier)
        else:
            self.embedder = PolicyAnalysisEmbedder(config, retry_handler=retry_handler)

        self._logger = logging.getLogger(self.__class__.__name__)
        self._logger.info("EmbeddingPolicyProducer initialized")

    # ========================================================================
    # DOCUMENT PROCESSING API
    # ========================================================================

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

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_ch
unk_count")
    def get_chunk_count(self, chunks: list[SemanticChunk]) -> int:
        """Get number of chunks"""
        return len(chunks)

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_ch
unk_text")
    def get_chunk_text(self, chunk: SemanticChunk) -> str:
        """Extract text from chunk"""
        return chunk["content"]

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_ch
unk_embedding")
    def get_chunk_embedding(self, chunk: SemanticChunk) -> NDArray[np.float32]:
        """Extract embedding from chunk"""
        return chunk["embedding"]

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_ch
unk_metadata")
    def get_chunk_metadata(self, chunk: SemanticChunk) -> dict[str, Any]:
        """Extract metadata from chunk"""
        return chunk["metadata"]
    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_ch
unk_pdq_context")
    def get_chunk_pdq_context(self, chunk: SemanticChunk) -> PDQIdentifier | None:
        """Extract P-D-Q context from chunk"""
        return chunk["pdq_context"]

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

    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 reranking"""
        return self.embedder.semantic_search(
            query, document_chunks, pdq_filter, use_reranking
        )

    def get_search_result_chunk(
        self, result: tuple[SemanticChunk, float]
    ) -> SemanticChunk:
        """Extract chunk from search result"""
        return result[0]

    def get_search_result_score(
        self, result: tuple[SemanticChunk, float]
    ) -> float:
        """Extract relevance score from search result"""
        return result[1]

    # ============================================================================
    # P-D-Q ANALYSIS API
    # ============================================================================

    def generate_pdq_report(
        self,
        document_chunks: list[SemanticChunk],
        target_pdq: PDQIdentifier
    ) -> dict[str, Any]:
        """Generate comprehensive analytical report for P-D-Q question"""
        return self.embedder.generate_pdq_report(document_chunks, target_pdq)

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_pd
q_evidence_count")
    def get_pdq_evidence_count(self, report: dict[str, Any]) -> int:
        """Extract evidence count from P-D-Q report"""
        return report.get("evidence_count", 0)

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_pd
q_numerical_evaluation")
    def get_pdq_numerical_evaluation(self, report: dict[str, Any]) -> dict[str, Any]:
        """Extract numerical evaluation from P-D-Q report"""
        return report.get("numerical_evaluation", {})

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_pd
q_evidence_passages")
    def get_pdq_evidence_passages(self, report: dict[str, Any]) -> list[dict[str, Any]]:
        """Extract evidence passages from P-D-Q report"""
        return report.get("evidence_passages", [])

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_pd
q_confidence")
    def get_pdq_confidence(self, report: dict[str, Any]) -> float:
        """Extract confidence from P-D-Q report"""
        return report.get("confidence", get_parameter_loader().get("saaaaaa.processing.emb
edding_policy.EmbeddingPolicyProducer.get_pdq_confidence").get("auto_param_L1744_40",
0.0))

    # ========================================================================
    # BAYESIAN NUMERICAL ANALYSIS API
    # ========================================================================

    def evaluate_numerical_consistency(
        self,
        chunks: list[SemanticChunk],
        pdq_context: PDQIdentifier
    ) -> BayesianEvaluation:
        """Evaluate numerical consistency with Bayesian analysis"""
        return self.embedder.evaluate_policy_numerical_consistency(
            chunks, pdq_context
        )

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_po
int_estimate")
    def get_point_estimate(self, evaluation: BayesianEvaluation) -> float:
        """Extract point estimate from Bayesian evaluation"""
        return evaluation["point_estimate"]

    def get_credible_interval(
        self, evaluation: BayesianEvaluation
    ) -> tuple[float, float]:
        """Extract 95% credible interval from Bayesian evaluation"""
        return evaluation["credible_interval_95"]

    def get_evidence_strength(
        self, evaluation: BayesianEvaluation
    ) -> Literal["weak", "moderate", "strong", "very_strong"]:
        """Extract evidence strength classification"""
        return evaluation["evidence_strength"]

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_nu
merical_coherence")
    def get_numerical_coherence(self, evaluation: BayesianEvaluation) -> float:
        """Extract numerical coherence score"""
        return evaluation["numerical_coherence"]

    # ========================================================================
    # POLICY COMPARISON API
    # ========================================================================

    def compare_policy_interventions(
        self,
        intervention_a_chunks: list[SemanticChunk],
        intervention_b_chunks: list[SemanticChunk],
        pdq_context: PDQIdentifier
    ) -> dict[str, Any]:
        """Bayesian comparison of two policy interventions"""
        return self.embedder.compare_policy_interventions(
            intervention_a_chunks, intervention_b_chunks, pdq_context
        )

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_co
mparison_probability")
    def get_comparison_probability(self, comparison: dict[str, Any]) -> float:
        """Extract probability that A is better than B"""
        return comparison.get("probability_a_better", get_parameter_loader().get("saaaaaa.
processing.embedding_policy.EmbeddingPolicyProducer.get_comparison_probability").get("auto
_param_L1800_54", 0.5))

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_co
mparison_bayes_factor")
    def get_comparison_bayes_factor(self, comparison: dict[str, Any]) -> float:
        """Extract Bayes factor from comparison"""
        return comparison.get("bayes_factor", get_parameter_loader().get("saaaaaa.processi
ng.embedding_policy.EmbeddingPolicyProducer.get_comparison_bayes_factor").get("auto_param_
L1805_46", 1.0))

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_co
mparison_difference_mean")
    def get_comparison_difference_mean(self, comparison: dict[str, Any]) -> float:
        """Extract mean difference from comparison"""
        return comparison.get("difference_mean", get_parameter_loader().get("saaaaaa.proce
ssing.embedding_policy.EmbeddingPolicyProducer.get_comparison_difference_mean").get("auto_
param_L1810_49", 0.0))

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

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_di
agnostics")
    def get_diagnostics(self) -> dict[str, Any]:
        """Get system diagnostics and performance metrics"""
        return self.embedder.get_diagnostics()

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_co
nfig")
    def get_config(self) -> PolicyEmbeddingConfig:
        """Get current configuration"""
        return self.embedder.config

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.list_p
olicy_domains")
    def list_policy_domains(self) -> list[PolicyDomain]:
        """List all policy domains"""
        return list(PolicyDomain)

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.list_a
nalytical_dimensions")
    def list_analytical_dimensions(self) -> list[AnalyticalDimension]:
        """List all analytical dimensions"""
        return list(AnalyticalDimension)

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_po
licy_domain_description")
    def get_policy_domain_description(self, domain: PolicyDomain) -> str:
        """Get description for policy domain"""
        return domain.value

    @calibrated_method("saaaaaa.processing.embedding_policy.EmbeddingPolicyProducer.get_an
alytical_dimension_description")
    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 ESTRATÉGICO 1: DERECHOS DE LAS MUJERES E IGUALDAD DE GÉNERO

    DIAGNÓSTICO
    El municipio presenta una brecha de género del 18.5% en participación laboral.
    Se identificaron 2,340 mujeres en situación de vulnerabilidad económica.
    El presupuesto asignado asciende a $450 millones para el cuatrienio.

    DISEÑO DE INTERVENCIÓN
    Se implementarán 3 programas de empoderamiento económico:
    - Programa de formación técnica: 500 beneficiarias
    - Microcréditos productivos: $280 millones
    - Fortalecimiento empresarial: 150 emprendimientos

    PRODUCTOS Y OUTPUTS
    Meta cuatrienio: reducir brecha de género al 12% (reducción del 35.1%)
    Indicador: Tasa de participación laboral femenina
    Línea base: 42.3% | Meta: 55.8%

    RESULTADOS ESPERADOS
    Incremento del 25% en ingresos promedio de beneficiarias
    Creación de 320 nuevos empleos formales para mujeres
    Sostenibilidad: 78% de emprendimientos activos a 2 años
    """

    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/saaaaaa/processing/factory.py =====
"""
Factory Layer for Processing Module I/O Operations

This module provides centralized I/O operations for the processing package,
implementing a clean separation between I/O and business logic following
the Ports and Adapters (Hexagonal Architecture) pattern.

All file I/O for the processing package should be handled through this factory.
"""

import hashlib
import json
import logging
from pathlib import Path
from typing import Any
from saaaaaa.core.calibration.decorators import calibrated_method

try:
    import pdfplumber
except ImportError:
    pdfplumber = None

logger = logging.getLogger(__name__)
```

```
# ============================================================================
# FILE I/O OPERATIONS
# ============================================================================

def load_json(file_path: str | Path) -> dict[str, Any]:
    """
    Load JSON data from file.

    Args:
        file_path: Path to JSON file

    Returns:
        Dict containing the loaded JSON data

    Raises:
        FileNotFoundError: If file doesn't exist
        json.JSONDecodeError: If file contains invalid JSON
    """
    file_path = Path(file_path)

    if not file_path.exists():
        raise FileNotFoundError(f"File not found: {file_path}")

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

    logger.info(f"Loaded JSON from {file_path}")
    return data

def save_json(data: dict[str, Any], file_path: str | Path, indent: int = 2) -> None:
    """
    Save data to JSON file with formatted output.

    Args:
        data: Dictionary to save
        file_path: Path to output JSON file
        indent: Indentation level for formatting
    """
    file_path = Path(file_path)
    file_path.parent.mkdir(parents=True, exist_ok=True)

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

    logger.info(f"Saved JSON to {file_path}")

def read_text_file(file_path: str | Path, encodings: list = None) -> str:
    """
    Read text file with automatic encoding detection.

    Args:
        file_path: Path to text file
        encodings: List of encodings to try (default: utf-8, latin-1, cp1252)

    Returns:
        String content of the file

    Raises:
        FileNotFoundError: If file doesn't exist
        UnicodeDecodeError: If file cannot be decoded with any encoding
    """
    if encodings is None:
        encodings = ["utf-8", "latin-1", "cp1252"]

    file_path = Path(file_path)

    if not file_path.exists():
        raise FileNotFoundError(f"File not found: {file_path}")
    last_error = None
    for encoding in encodings:
        try:
            with open(file_path, encoding=encoding) as f:
                content = f.read()
            logger.debug(f"Successfully read {file_path} with {encoding}")
            return content
        except (UnicodeDecodeError, UnicodeError) as e:
            last_error = e
            continue

    raise UnicodeDecodeError(
        "utf-8", b"", 0, 0,
        f"Could not decode {file_path} with any of: {encodings}. Last error: {last_error}"
    )

def write_text_file(content: str, file_path: str | Path) -> None:
    """
    Write text content to file with UTF-8 encoding.

    Args:
        content: Text content to write
        file_path: Path to output file
    """
    file_path = Path(file_path)
    file_path.parent.mkdir(parents=True, exist_ok=True)

    with open(file_path, "w", encoding="utf-8") as f:
        f.write(content)

    logger.info(f"Written {len(content)} characters to {file_path}")

def calculate_file_hash(file_path: str | Path) -> str:
    """
    Calculate SHA-256 hash of a file for traceability.

    Args:
        file_path: Path to file

    Returns:
        Hexadecimal string representation of the file's SHA-256 hash
    """
    file_path = Path(file_path)
    sha256_hash = hashlib.sha256()

    with open(file_path, "rb") as f:
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)

    return sha256_hash.hexdigest()

# ============================================================================
# PDF OPERATIONS
# ============================================================================

def extract_pdf_text_all_pages(file_path: str | Path) -> str:
    """
    Extract all text from a PDF file.

    Args:
        file_path: Path to PDF file

    Returns:
        Concatenated text from all pages

    Raises:
        ImportError: If pdfplumber is not installed
        FileNotFoundError: If file doesn't exist
    """
    if pdfplumber is None:
        raise ImportError("pdfplumber is required for PDF operations. Install with: pip
install pdfplumber")

    file_path = Path(file_path)

    if not file_path.exists():
        raise FileNotFoundError(f"PDF file not found: {file_path}")

    all_text = []

    with pdfplumber.open(file_path) as pdf:
        for page_num, page in enumerate(pdf.pages, start=1):
            try:
                text = page.extract_text() or ""
                if text.strip():
                    all_text.append(f"\n--- Página {page_num} ---\n")
                    all_text.append(text)
            except Exception as e:
                logger.warning(f"Error extracting page {page_num}: {e}")
                continue

    result = "\n".join(all_text)
    logger.info(f"Extracted {len(result)} characters from {file_path}")
    return result

def extract_pdf_text_single_page(file_path: str | Path, page_num: int, total_pages: int =
None) -> str:
    """
    Extract text from a single page of a PDF.

    Args:
        file_path: Path to PDF file
        page_num: Page number to extract (1-indexed)
        total_pages: Total number of pages (optional, for validation)

    Returns:
        Text content of the specified page

    Raises:
        ImportError: If pdfplumber is not installed
        FileNotFoundError: If file doesn't exist
        ValueError: If page number is out of range
    """
    if pdfplumber is None:
        raise ImportError("pdfplumber is required for PDF operations. Install with: pip
install pdfplumber")

    file_path = Path(file_path)

    if not file_path.exists():
        raise FileNotFoundError(f"PDF file not found: {file_path}")

    with pdfplumber.open(file_path) as pdf:
        if total_pages and (page_num < 1 or page_num > total_pages):
            raise ValueError(f"Page {page_num} out of range (1-{total_pages})")

        if page_num < 1 or page_num > len(pdf.pages):
            raise ValueError(f"Page {page_num} out of range (1-{len(pdf.pages)})")

        text = pdf.pages[page_num - 1].extract_text() or ""
        return text

def get_pdf_page_count(file_path: str | Path) -> int:
    """
    Get the number of pages in a PDF file.
    Args:
        file_path: Path to PDF file

    Returns:
        Number of pages in the PDF

    Raises:
        ImportError: If pdfplumber is not installed
        FileNotFoundError: If file doesn't exist
    """
    if pdfplumber is None:
        raise ImportError("pdfplumber is required for PDF operations. Install with: pip
install pdfplumber")

    file_path = Path(file_path)

    if not file_path.exists():
        raise FileNotFoundError(f"PDF file not found: {file_path}")

    with pdfplumber.open(file_path) as pdf:
        return len(pdf.pages)

===== FILE: src/saaaaaa/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 saaaaaa.utils.runtime_error_fixes import ensure_list_return

try:
    from saaaaaa.analysis.contradiction_deteccion import (
        BayesianConfidenceCalculator,
        PolicyContradictionDetector,
        TemporalLogicVerifier,
    )
    from saaaaaa.analysis.contradiction_deteccion import (
        PolicyDimension as ContradictionPolicyDimension,
    )
    CONTRADICTION_MODULE_AVAILABLE = True
except Exception as import_error:  # pragma: no cover - safety net for heavy deps
    CONTRADICTION_MODULE_AVAILABLE = False

    # In production/CI, require the module to be available
    import os
    if os.getenv('REQUIRE_CONTRADICTION_MODULE', '').lower() in ('true', '1', 'yes'):
        raise ImportError(f"Contradiction detection module is required but not available: 
{import_error}")

    logger = logging.getLogger(__name__)
    logger.warning(
        "Falling back to lightweight contradiction components due to import error: %s",
        import_error,
    )

    class BayesianConfidenceCalculator:  # type: ignore[misc]
        """Fallback Bayesian calculator when advanced module is unavailable."""

        def __init__(self) -> None:
            self.prior_alpha = get_parameter_loader().get("saaaaaa.processing.policy_proce
ssor.BayesianConfidenceCalculator.__init__").get("auto_param_L64_31", 1.0)
            self.prior_beta = get_parameter_loader().get("saaaaaa.processing.policy_proces
sor.BayesianConfidenceCalculator.__init__").get("auto_param_L65_30", 1.0)

        def calculate_posterior(
            self, evidence_strength: float, observations: int, domain_weight: float = get_
parameter_loader().get("saaaaaa.processing.policy_processor.BayesianConfidenceCalculator._
_init__").get("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 TemporalLogicVerifier:  # type: ignore[misc]
        """Fallback temporal verifier providing graceful degradation."""

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

    class _FallbackContradictionDetector:
        def detect(
            self,
            text: str,
            plan_name: str = "PDM",
            dimension: Any = None,
        ) -> dict[str, Any]:
            return {
                "plan_name": plan_name,
                "dimension": getattr(dimension, "value", "unknown"),
                "contradictions": [],
                "total_contradictions": 0,
                "high_severity_count": 0,
                "coherence_metrics": {},
                "recommendations": [],
                "knowledge_graph_stats": {"nodes": 0, "edges": 0, "components": 0},
            }

        @calibrated_method("saaaaaa.processing.policy_processor._FallbackContradictionDete
ctor._extract_policy_statements")
        def _extract_policy_statements(self, text: str, dimension: Any) -> list[Any]:
            return []
    PolicyContradictionDetector = _FallbackContradictionDetector  # type: ignore[misc]

    class ContradictionPolicyDimension(Enum):  # type: ignore[misc]
        DIAGNOSTICO = "diagnóstico"
        ESTRATEGICO = "estratégico"
        PROGRAMATICO = "programático"
        FINANCIERO = "plan plurianual de inversiones"
        SEGUIMIENTO = "seguimiento y evaluación"
        TERRITORIAL = "ordenamiento territorial"

from saaaaaa.analysis.Analyzer_one import (
    DocumentProcessor,
    MunicipalAnalyzer,
    MunicipalOntology,
    PerformanceAnalyzer,
    SemanticAnalyzer,
)
from saaaaaa.analysis.financiero_viabilidad_tablas import PDETAnalysisException,
QualityScore
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

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

# ============================================================================
# CAUSAL DIMENSION TAXONOMY (DECALOGO Framework)
# ============================================================================

class CausalDimension(Enum):
    """Six-dimensional causal framework taxonomy aligned with DECALOGO."""

    D1_INSUMOS = "d1_insumos"
    D2_ACTIVIDADES = "d2_actividades"
    D3_PRODUCTOS = "d3_productos"
    D4_RESULTADOS = "d4_resultados"
    D5_IMPACTOS = "d5_impactos"
    D6_CAUSALIDAD = "d6_causalidad"

# ============================================================================
# ENHANCED PATTERN LIBRARY WITH SEMANTIC HIERARCHIES
# ============================================================================

CAUSAL_PATTERN_TAXONOMY: dict[CausalDimension, dict[str, list[str]]] = {
    CausalDimension.D1_INSUMOS: {
        "diagnostico_cuantitativo": [
            r"\b(?:diagn[óo]stico\s+(?:cuantitativo|estad[íi]stico|situacional))\b",
            r"\b(?:an[áa]lisis\s+(?:de\s+)?(?:brecha|situaci[óo]n\s+actual))\b",
            r"\b(?:caracterizaci[óo]n\s+(?:territorial|poblacional|sectorial))\b",
        ],
        "lineas_base_temporales": [
            r"\b(?:l[íi]nea(?:s)?\s+(?:de\s+)?base)\b",
            r"\b(?:valor(?:es)?\s+inicial(?:es)?)\b",
            r"\b(?:serie(?:s)?\s+(?:hist[óo]rica(?:s)?|temporal(?:es)?))\b",
            r"\b(?:medici[óo]n\s+(?:de\s+)?referencia)\b",
        ],
        "recursos_programaticos": [
            r"\b(?:presupuesto\s+(?:plurianual|de\s+inversi[óo]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[áa]tica))\b",
        ],
        "capacidad_institucional": [
            r"\b(?:capacidad(?:es)?\s+(?:institucional(?:es)?|t[ée]cnica(?:s)?))\b",
            r"\b(?:talento\s+humano\s+(?:disponible|requerido))\b",
            r"\b(?:gobernanza\s+(?:de\s+)?(?:datos|informaci[óo]n))\b",
            r"\b(?:brechas?\s+(?:de\s+)?implementaci[óo]n)\b",
        ],
    },
    CausalDimension.D2_ACTIVIDADES: {
        "formalizacion_actividades": [
            r"\b(?:plan\s+de\s+acci[óo]n\s+detallado)\b",
            r"\b(?:matriz\s+de\s+(?:actividades|intervenciones))\b",
            r"\b(?:cronograma\s+(?:de\s+)?ejecuci[óo]n)\b",
            r"\b(?:responsables?\s+(?:designados?|identificados?))\b",
        ],
        "mecanismo_causal": [
            r"\b(?:mecanismo(?:s)?\s+causal(?:es)?)\b",
            r"\b(?:teor[íi]a\s+(?:de\s+)?intervenci[óo]n)\b",
            r"\b(?:cadena\s+(?:de\s+)?causaci[óo]n)\b",
            r"\b(?:v[íi]nculo(?:s)?\s+explicativo(?:s)?)\b",
        ],
        "poblacion_objetivo": [
            r"\b(?:poblaci[óo]n\s+(?:diana|objetivo|beneficiaria))\b",
            r"\b(?:criterios?\s+de\s+focalizaci[óo]n)\b",
            r"\b(?:segmentaci[óo]n\s+(?:territorial|poblacional))\b",
        ],
        "dosificacion_intervencion": [
            r"\b(?:dosificaci[óo]n\s+(?:de\s+)?(?:la\s+)?intervenci[óo]n)\b",
            r"\b(?:intensidad\s+(?:de\s+)?tratamiento)\b",
            r"\b(?:duraci[óo]n\s+(?:de\s+)?exposici[óo]n)\b",
        ],
    },
    CausalDimension.D3_PRODUCTOS: {
        "indicadores_producto": [
            r"\b(?:indicador(?:es)?\s+de\s+(?:producto|output|gesti[óo]n))\b",
            r"\b(?:entregables?\s+verificables?)\b",
            r"\b(?:metas?\s+(?:de\s+)?producto)\b",
        ],
        "verificabilidad": [
            r"\b(?:f[óo]rmula\s+(?:de\s+)?(?:c[áa]lculo|medici[óo]n))\b",
            r"\b(?:fuente(?:s)?\s+(?:de\s+)?verificaci[óo]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[ée]trica(?:s)?)\s+de\s+(?:resultado|outcome))\b",
            r"\b(?:criterios?\s+de\s+[ée]xito)\b",
            r"\b(?:umbral(?:es)?\s+de\s+desempe[ñn]o)\b",
        ],
        "encadenamiento_causal": [
            r"\b(?:encadenamiento\s+(?:causal|l[óo]gico))\b",
            r"\b(?:ruta(?:s)?\s+cr[íi]tica(?:s)?)\b",
            r"\b(?:dependencias?\s+causales?)\b",
        ],
        "ventana_maduracion": [
            r"\b(?:ventana\s+de\s+maduraci[óo]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[óo]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[óo]n\s+(?:estructural|sistémica))\b",
        ],
        "rutas_transmision": [
            r"\b(?:ruta(?:s)?\s+de\s+transmisi[óo]n)\b",
            r"\b(?:canales?\s+(?:de\s+)?(?:impacto|propagaci[óo]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[íi]ntesis))\b",
            r"\b(?:medidas?\s+(?:indirectas?|aproximadas?))\b",
        ],
        "alineacion_marcos": [
            r"\b(?:alineaci[óo]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[íi]a\s+de(?:l)?\s+cambio)\b",
            r"\b(?:modelo\s+l[óo]gico\s+(?:integrado|completo))\b",
            r"\b(?:marco\s+causal\s+(?:expl[íi]cito|formalizado))\b",
        ],
        "diagrama_causal": [
            r"\b(?:diagrama\s+(?:causal|DAG|de\s+flujo))\b",
            r"\b(?:representaci[óo]n\s+gr[áa]fica\s+causal)\b",
            r"\b(?:mapa\s+(?:de\s+)?relaciones?)\b",
        ],
        "supuestos_verificables": [
            r"\b(?:supuestos?\s+(?:verificables?|cr[íi]ticos?))\b",
            r"\b(?:hip[óo]tesis\s+(?:causales?|comprobables?))\b",
            r"\b(?:condiciones?\s+(?:necesarias?|suficientes?))\b",
        ],
        "mediadores_moderadores": [
            r"\b(?:mediador(?:es)?|moderador(?:es)?)\b",
            r"\b(?:variables?\s+(?:intermedias?|mediadoras?|moderadoras?))\b",
        ],
        "validacion_logica": [
            r"\b(?:validaci[óo]n\s+(?:l[óo]gica|emp[íi]rica))\b",
            r"\b(?:pruebas?\s+(?:de\s+)?consistencia)\b",
            r"\b(?:auditor[íi]a\s+causal)\b",
        ],
        "sistema_seguimiento": [
            r"\b(?:sistema\s+de\s+(?:seguimiento|monitoreo))\b",
            r"\b(?:tablero\s+de\s+(?:control|indicadores))\b",
            r"\b(?:evaluaci[óo]n\s+(?:continua|peri[óo]dica))\b",
        ],
    },
}

# ============================================================================
# CONFIGURATION ARCHITECTURE
# ============================================================================

@dataclass(frozen=True)
class ProcessorConfig:
    """Immutable configuration for policy plan processing."""

    preserve_document_structure: bool = True
    enable_semantic_tagging: bool = True
    confidence_threshold: float = get_parameter_loader().get("saaaaaa.processing.policy_pr
ocessor._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L301_3
4", 0.65)
    context_window_chars: int = 400
    max_evidence_per_pattern: int = 5
    enable_bayesian_scoring: bool = True
    utf8_normalization_form: str = "NFC"

    # Advanced controls
    entropy_weight: float = get_parameter_loader().get("saaaaaa.processing.policy_processo
r._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L308_28",
0.3)
    proximity_decay_rate: float = get_parameter_loader().get("saaaaaa.processing.policy_pr
ocessor._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L309_3
4", 0.15)
    min_sentence_length: int = 20
    max_sentence_length: int = 500
    bayesian_prior_confidence: float = get_parameter_loader().get("saaaaaa.processing.poli
cy_processor._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L
312_39", 0.5)
    bayesian_entropy_weight: float = get_parameter_loader().get("saaaaaa.processing.policy
_processor._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L31
3_37", 0.3)
    minimum_dimension_scores: dict[str, float] = field(
        default_factory=lambda: {
            "D1": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L316_18", 0.50),
            "D2": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L317_18", 0.50),
            "D3": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L318_18", 0.50),
            "D4": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L319_18", 0.50),
            "D5": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L320_18", 0.50),
            "D6": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fallbac
kContradictionDetector._extract_policy_statements").get("auto_param_L321_18", 0.50),
        }
    )
    critical_dimension_overrides: dict[str, float] = field(
        default_factory=lambda: {"D1": get_parameter_loader().get("saaaaaa.processing.poli
cy_processor._FallbackContradictionDetector._extract_policy_statements").get("auto_param_L
325_39", 0.55), "D6": get_parameter_loader().get("saaaaaa.processing.policy_processor._Fal
lbackContradictionDetector._extract_policy_statements").get("auto_param_L325_51", 0.55)}
    )
    differential_focus_indicators: tuple[str, ...] = (
        "enfoque diferencial",
        "enfoque de género",
        "mujeres rurales",
        "población víctima",
        "firmantes del acuerdo",
        "comunidades indígenas",
        "población LGBTIQ+",
        "juventud rural",
        "comunidades ribereñas",
    )
    adaptability_indicators: tuple[str, ...] = (
        "mecanismo de ajuste",
        "retroalimentación",
        "aprendizaje",
        "monitoreo adaptativo",
        "ciclo de mejora",
        "sistema de alerta temprana",
        "evaluación continua",
    )

    LEGACY_PARAM_MAP: ClassVar[dict[str, str]] = {
        "keep_structure": "preserve_document_structure",
        "tag_elements": "enable_semantic_tagging",
        "threshold": "confidence_threshold",
    }
    @classmethod
    def from_legacy(cls, **kwargs: Any) -> "ProcessorConfig":
        """Construct configuration from legacy parameter names."""
        normalized = {}
        for key, value in kwargs.items():
            canonical = cls.LEGACY_PARAM_MAP.get(key, key)
            normalized[canonical] = value
        return cls(**normalized)

    @calibrated_method("saaaaaa.processing.policy_processor.ProcessorConfig.validate")
    def validate(self) -> None:
        """Validate configuration parameters."""
        if not get_parameter_loader().get("saaaaaa.processing.policy_processor.ProcessorCo
nfig.validate").get("auto_param_L366_15", 0.0) <= self.confidence_threshold <= get_paramet
er_loader().get("saaaaaa.processing.policy_processor.ProcessorConfig.validate").get("auto_
param_L366_51", 1.0):
            raise ValueError("confidence_threshold must be in [0, 1]")
        if self.context_window_chars < 100:
            raise ValueError("context_window_chars must be >= 100")
        if self.entropy_weight < 0 or self.entropy_weight > 1:
            raise ValueError("entropy_weight must be in [0, 1]")
        if not get_parameter_loader().get("saaaaaa.processing.policy_processor.ProcessorCo
nfig.validate").get("auto_param_L372_15", 0.0) <= self.bayesian_prior_confidence <= get_pa
rameter_loader().get("saaaaaa.processing.policy_processor.ProcessorConfig.validate").get("
auto_param_L372_56", 1.0):
            raise ValueError("bayesian_prior_confidence must be in [0, 1]")
        if not get_parameter_loader().get("saaaaaa.processing.policy_processor.ProcessorCo
nfig.validate").get("auto_param_L374_15", 0.0) <= self.bayesian_entropy_weight <= get_para
meter_loader().get("saaaaaa.processing.policy_processor.ProcessorConfig.validate").get("au
to_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 get_parameter_loader().get("saaaaaa.processing.policy_processor.Process
orConfig.validate").get("auto_param_L377_19", 0.0) <= threshold <= get_parameter_loader().
get("saaaaaa.processing.policy_processor.ProcessorConfig.validate").get("auto_param_L377_3
9", 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 get_parameter_loader().get("saaaaaa.processing.policy_processor.Process
orConfig.validate").get("auto_param_L382_19", 0.0) <= threshold <= get_parameter_loader().
get("saaaaaa.processing.policy_processor.ProcessorConfig.validate").get("auto_param_L382_3
9", 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 = get_parameter_loader().get("saaaaaa.processing.policy_pr
ocessor.ProcessorConfig.validate").get("auto_param_L401_34", 0.5),
        entropy_weight: float = get_parameter_loader().get("saaaaaa.processing.policy_proc
essor.ProcessorConfig.validate").get("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 = get_parameter_loader().get("saaaaaa.processing.policy_p
rocessor.ProcessorConfig.validate").get("auto_param_L411_35", 0.02)
        self.duplicate_gamma: float = get_parameter_loader().get("saaaaaa.processing.polic
y_processor.ProcessorConfig.validate").get("auto_param_L412_38", 1.0)
        self.cross_type_floor: float = get_parameter_loader().get("saaaaaa.processing.poli
cy_processor.ProcessorConfig.validate").get("auto_param_L413_39", 0.0)
        self.source_quality_weights: dict[str, float] = {}
        self.sector_multipliers: dict[str, float] = {}
        self.sector_default: float = get_parameter_loader().get("saaaaaa.processing.policy
_processor.ProcessorConfig.validate").get("auto_param_L416_37", 1.0)
        self.municipio_multipliers: dict[str, float] = {}
        self.municipio_default: float = get_parameter_loader().get("saaaaaa.processing.pol
icy_processor.ProcessorConfig.validate").get("auto_param_L418_40", 1.0)

        self._configure_from_calibration()

    @calibrated_method("saaaaaa.processing.policy_processor.BayesianEvidenceScorer._config
ure_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, get_parameter_loader().get(
"saaaaaa.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration").
get("auto_param_L435_63", 0.0)), get_parameter_loader().get("saaaaaa.processing.policy_pro
cessor.BayesianEvidenceScorer._configure_from_calibration").get("auto_param_L435_69",
0.45))
                self.duplicate_gamma = max(get_parameter_loader().get("saaaaaa.processing.
policy_processor.BayesianEvidenceScorer._configure_from_calibration").get("auto_param_L436
_43", 0.0), self.duplicate_gamma)
                self.cross_type_floor = max(get_parameter_loader().get("saaaaaa.processing
.policy_processor.BayesianEvidenceScorer._configure_from_calibration").get("auto_param_L43
7_44", 0.0), min(get_parameter_loader().get("saaaaaa.processing.policy_processor.BayesianE
videnceScorer._configure_from_calibration").get("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", get
_parameter_loader().get("saaaaaa.processing.policy_processor.BayesianEvidenceScorer._confi
gure_from_calibration").get("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", get_parameter_loader().get("saaaaaa.proces
sing.policy_processor.BayesianEvidenceScorer._configure_from_calibration").get("auto_param
_L454_93", 1.0)))

    def compute_evidence_score(
        self,
        matches: list[str],
        total_corpus_size: int,
        pattern_specificity: float = get_parameter_loader().get("saaaaaa.processing.policy
_processor.BayesianEvidenceScorer._configure_from_calibration").get("auto_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

        Returns:
            Calibrated confidence score in [0, 1]
        """
        if not matches:
            return get_parameter_loader().get("saaaaaa.processing.policy_processor.Bayesia
nEvidenceScorer._configure_from_calibration").get("auto_param_L476_19", 0.0)

        # Term frequency normalization
        tf = len(matches) / max(1, total_corpus_size / 1000)
        if self.cross_type_floor:
            tf = max(self.cross_type_floor, tf)

        # Entropy-based diversity penalty
        match_lengths = np.array([len(m) for m in matches])
        entropy = self._calculate_shannon_entropy(match_lengths)

        # Bayesian update
        clip_low = self.epsilon_clip
        clip_high = get_parameter_loader().get("saaaaaa.processing.policy_processor.Bayesi
anEvidenceScorer._configure_from_calibration").get("auto_param_L489_20", 1.0) -
self.epsilon_clip
        pattern_specificity = max(clip_low, min(clip_high, pattern_specificity))

        likelihood = min(get_parameter_loader().get("saaaaaa.processing.policy_processor.B
ayesianEvidenceScorer._configure_from_calibration").get("auto_param_L492_25", 1.0), tf *
pattern_specificity)
        posterior = (likelihood * self.prior) / (
            (likelihood * self.prior) + ((1 - likelihood) * (1 - self.prior))
        )

        # Entropy-weighted adjustment
        final_score = (1 - self.entropy_weight) * posterior + self.entropy_weight * (
            1 - entropy
        )

        # Apply duplicate penalty if provided by caller
        if kwargs.get("duplicate_penalty"):
            final_score *= self.duplicate_gamma
        # Apply source quality weighting
        if self.source_quality_weights:
            source_quality = kwargs.get("source_quality")
            if source_quality is not None:
                weight = self._lookup_weight(self.source_quality_weights, source_quality,
default=get_parameter_loader().get("saaaaaa.processing.policy_processor.BayesianEvidenceSc
orer._configure_from_calibration").get("auto_param_L510_98", 1.0))
                final_score *= weight

        # Context multipliers (sector / municipality)
        sector = kwargs.get("sector") or kwargs.get("policy_sector")
        if self.sector_multipliers:
            final_score *= self._lookup_weight(self.sector_multipliers, sector,
default=self.sector_default)

        municipio = kwargs.get("municipio_tamano") or kwargs.get("municipio_size")
        if self.municipio_multipliers:
            final_score *= self._lookup_weight(self.municipio_multipliers, municipio,
default=self.municipio_default)

        return np.clip(final_score, get_parameter_loader().get("saaaaaa.processing.policy_
processor.BayesianEvidenceScorer._configure_from_calibration").get("auto_param_L522_36",
0.0), get_parameter_loader().get("saaaaaa.processing.policy_processor.BayesianEvidenceScor
er._configure_from_calibration").get("auto_param_L522_41", 1.0))

    @staticmethod
    def _calculate_shannon_entropy(values: np.ndarray, **kwargs: Any) -> float:
        """Calculate normalized Shannon entropy for value distribution.

        Args:
            values: Array of numerical values
            **kwargs: Additional optional parameters for compatibility

        Returns:
            Normalized Shannon entropy
        """
        if len(values) < 2:
            return get_parameter_loader().get("saaaaaa.processing.policy_processor.Bayesia
nEvidenceScorer._configure_from_calibration").get("auto_param_L536_19", 0.0)

        # Discrete probability distribution
        hist, _ = np.histogram(values, bins=min(10, len(values)))
        prob = hist / hist.sum()
        prob = prob[prob > 0]  # Remove zeros

        entropy = -np.sum(prob * np.log2(prob))
        max_entropy = np.log2(len(prob)) if len(prob) > 1 else get_parameter_loader().get(
"saaaaaa.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration").
get("auto_param_L544_63", 1.0)

        return entropy / max_entropy if max_entropy > 0 else get_parameter_loader().get("s
aaaaaa.processing.policy_processor.BayesianEvidenceScorer._configure_from_calibration").ge
t("auto_param_L546_61", 0.0)

    @staticmethod
    def _lookup_weight(mapping: dict[str, float], key: Any, default: float = get_parameter
_loader().get("saaaaaa.processing.policy_processor.BayesianEvidenceScorer._configure_from_
calibration").get("auto_param_L549_77", 1.0)) -> float:
        if not mapping:
            return default
        if key is None:
            return mapping.get("default", default)
        if isinstance(key, str):
            direct = mapping.get(key)
            if direct is not None:
                return direct
            lowered = key.lower()
            for candidate, value in mapping.items():
                if isinstance(candidate, str) and candidate.lower() == lowered:
                    return value
        return mapping.get("default", default)

# ============================================================================
# ADVANCED TEXT PROCESSOR
# ============================================================================

class PolicyTextProcessor:
    """
    Industrial-grade text processing with multi-scale segmentation and
    coherence-preserving normalization for policy document analysis.
    """

    def __init__(self, config: ProcessorConfig, *, calibration: dict[str, Any] | None =
None) -> None:
        self.config = config
        self.calibration = calibration or {}
        self._compiled_patterns: dict[str, re.Pattern] = {}
        self._sentence_boundaries = re.compile(
            r"(?<=[.!?])\s+(?=[A-ZÁÉÍÓÚÑ])|(?<=\n\n)"
        )

    @calibrated_method("saaaaaa.processing.policy_processor.PolicyTextProcessor.normalize_
unicode")
    def normalize_unicode(self, text: str) -> str:
        """Apply canonical Unicode normalization (NFC/NFKC)."""
        return unicodedata.normalize(self.config.utf8_normalization_form, text)

    @calibrated_method("saaaaaa.processing.policy_processor.PolicyTextProcessor.segment_in
to_sentences")
    def segment_into_sentences(self, text: str, **kwargs: Any) -> list[str]:
        """
        Segment text into sentences with context-aware boundary detection.
        Handles abbreviations, numerical lists, and Colombian naming conventions.

        Args:
            text: Input text to segment
            **kwargs: Additional optional parameters for compatibility

        Returns:
            List of sentence strings
        """
        # Protect common abbreviations
        protected = text
        protected = re.sub(r"\bDr\.", "Dr___", protected)
        protected = re.sub(r"\bSr\.", "Sr___", protected)
        protected = re.sub(r"\bart\.", "art___", protected)
        protected = re.sub(r"\bInc\.", "Inc___", protected)

        sentences = self._sentence_boundaries.split(protected)

        # Restore protected patterns
        sentences = [s.replace("___", ".") for s in sentences]

        # Filter by length constraints
        return [
            s.strip()
            for s in sentences
            if self.config.min_sentence_length
            <= len(s.strip())
            <= self.config.max_sentence_length
        ]

    def extract_contextual_window(
        self, text: str, match_position: int, window_size: int
    ) -> str:
        """Extract semantically coherent context window around a match."""
        start = max(0, match_position - window_size // 2)
        end = min(len(text), match_position + window_size // 2)

        # Expand to sentence boundaries
        while start > 0 and text[start] not in ".!?\n":
            start -= 1
        while end < len(text) and text[end] not in ".!?\n":
            end += 1

        return text[start:end].strip()

    @lru_cache(maxsize=256)
    @calibrated_method("saaaaaa.processing.policy_processor.PolicyTextProcessor.compile_pa
ttern")
    def compile_pattern(self, pattern_str: str) -> re.Pattern:
        """Cache and compile regex patterns for performance."""
        return re.compile(pattern_str, re.IGNORECASE | re.UNICODE)

# ============================================================================
# CORE INDUSTRIAL PROCESSOR
# ============================================================================

@dataclass
class EvidenceBundle:
    """Structured evidence container with provenance and confidence metadata."""

    dimension: CausalDimension
    category: str
    matches: list[str] = field(default_factory=list)
    confidence: float = get_parameter_loader().get("saaaaaa.processing.policy_processor.Po
licyTextProcessor.compile_pattern").get("auto_param_L653_24", 0.0)
    context_windows: list[str] = field(default_factory=list)
    match_positions: list[int] = field(default_factory=list)

    @calibrated_method("saaaaaa.processing.policy_processor.EvidenceBundle.to_dict")
    def to_dict(self) -> dict[str, Any]:
        return {
            "dimension": self.dimension.value,
            "category": self.category,
            "match_count": len(self.matches),
            "confidence": round(self.confidence, 4),
            "evidence_samples": self.matches[:3],
            "context_preview": self.context_windows[:2],
        }

class IndustrialPolicyProcessor:
    """
    State-of-the-art policy plan processor implementing rigorous causal
    framework analysis with Bayesian evidence scoring and graph-theoretic
    validation for Colombian local development plans.

    This processor provides core analysis capabilities for policy documents.

    DEPRECATION NOTE: The questionnaire_path parameter is deprecated.
    Modern pipelines use SPC (Smart Policy Chunks) ingestion which handles
    questionnaire integration separately.
    """

    def __init__(
        self,
        config: ProcessorConfig | None = None,
        questionnaire_path: Path | None = None,  # DEPRECATED: Kept for API compatibility
only
        *,
        ontology: MunicipalOntology | None = None,
        semantic_analyzer: SemanticAnalyzer | None = None,
        performance_analyzer: PerformanceAnalyzer | None = None,
        contradiction_detector: Optional["PolicyContradictionDetector"] = None,
        temporal_verifier: TemporalLogicVerifier | None = None,
        confidence_calculator: BayesianConfidenceCalculator | None = None,
        municipal_analyzer: MunicipalAnalyzer | 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,
        )

        self.ontology = ontology or MunicipalOntology()
        self.semantic_analyzer = semantic_analyzer or SemanticAnalyzer(self.ontology)
        self.performance_analyzer = performance_analyzer or
PerformanceAnalyzer(self.ontology)
        self.contradiction_detector = contradiction_detector or
PolicyContradictionDetector()
        self.temporal_verifier = temporal_verifier or TemporalLogicVerifier()
        self.confidence_calculator = confidence_calculator or
BayesianConfidenceCalculator()
        self.municipal_analyzer = municipal_analyzer or MunicipalAnalyzer()

        # LEGACY: Questionnaire loading removed - this component is deprecated
        # Modern SPC pipeline handles questionnaire injection separately
        self.questionnaire_file_path = None
        self.questionnaire_data = {"questions": []}  # Empty stub for backward
compatibility

        # Compile pattern taxonomy
        self._pattern_registry = self._compile_pattern_registry()

        # Policy point keyword extraction
        self.point_patterns: dict[str, re.Pattern] = {}
        self._build_point_patterns()

        # Processing statistics
        self.statistics: dict[str, Any] = defaultdict(int)

    @calibrated_method("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor._loa
d_questionnaire")
    def _load_questionnaire(self) -> dict[str, Any]:
        """
        LEGACY: Questionnaire loading disabled.

        This method is kept for backward compatibility but returns empty data.
        Modern SPC pipeline handles questionnaire injection separately.
        """
        logger.warning(
            "IndustrialPolicyProcessor._load_questionnaire called but questionnaire "
            "loading is disabled. This is a legacy component. Use SPC ingestion instead."
        )
        return {"questions": []}

    @calibrated_method("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor._com
pile_pattern_registry")
```

```
    def _compile_pattern_registry(self) -> dict[CausalDimension, dict[str,
list[re.Pattern]]]:
        """Compile all causal patterns into efficient regex objects."""
        registry = {}
        for dimension, categories in CAUSAL_PATTERN_TAXONOMY.items():
            registry[dimension] = {}
            for category, patterns in categories.items():
                registry[dimension][category] = [
                    self.text_processor.compile_pattern(p) for p in patterns
                ]
        return registry

    @calibrated_method("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor._bui
ld_point_patterns")
    def _build_point_patterns(self) -> None:
        """
        LEGACY: Pattern building from questionnaire disabled.

        This method is kept for backward compatibility but does nothing.
        Modern SPC pipeline handles question-aware chunking separately.
        """
        questions = self.questionnaire_data.get("questions", [])

        if not questions:
            logger.info(
                "No questionnaire questions available. "
                "This is expected for legacy IndustrialPolicyProcessor. "
                "Use SPC ingestion for question-aware analysis."
            )
            return

        # Legacy path (should not be reached in modern pipeline)
        point_keywords: dict[str, set[str]] = defaultdict(set)

        for question in questions:
            point_code = question.get("point_code")
            if not point_code:
                continue

            # Extract title keywords
            title = question.get("point_title", "").lower()
            if title:
                point_keywords[point_code].add(title)

            # Extract hint keywords (cleaned)
            for hint in question.get("hints", []):
                cleaned = re.sub(r"[()]", "", hint).strip().lower()
                if len(cleaned) > 3:
                    point_keywords[point_code].add(cleaned)

        # Compile into optimized regex patterns
        for point_code, keywords in point_keywords.items():
            # Sort by length (prioritize longer phrases)
            sorted_kw = sorted(keywords, key=len, reverse=True)
            pattern_str = "|".join(rf"\b{re.escape(kw)}\b" for kw in sorted_kw if kw)
            self.point_patterns[point_code] = re.compile(pattern_str, re.IGNORECASE)

        logger.info(f"Compiled patterns for {len(self.point_patterns)} policy points")

    @calibrated_method("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.proc
ess")
    def process(self, raw_text: str, **kwargs: Any) -> dict[str, Any]:
        """
        Execute comprehensive policy plan analysis.

        Args:
            raw_text: Sanitized policy document text
            **kwargs: Additional optional parameters (e.g., text, sentences, tables) for
```

```
compatibility

        Returns:
            Structured analysis results with evidence bundles and confidence scores
        """
        if not raw_text or len(raw_text) < 100:
            logger.warning("Input text too short for analysis")
            return self._empty_result()

        # Normalize and segment
        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

        Returns:
            Computed confidence score
        """
        confidence = self.scorer.compute_evidence_score(
            matches, text_length, pattern_specificity=pattern_specificity
        )
        return confidence
    def _construct_evidence_bundle(
        self,
        dimension: CausalDimension,
        category: str,
        matches: list[str],
        positions: list[int],
        confidence: float,
        **kwargs: Any
    ) -> dict[str, Any]:
        """
        Assemble evidence bundle from matched patterns and computed confidence.

        Args:
            dimension: Causal dimension classification
            category: Specific category within dimension
            matches: List of matched pattern strings
            positions: List of match positions in text
            confidence: Computed confidence score
            **kwargs: Additional optional parameters for compatibility

        Returns:
            Serialized evidence bundle dictionary
        """
        bundle = EvidenceBundle(
            dimension=dimension,
            category=category,
            matches=matches[: self.config.max_evidence_per_pattern],
            confidence=confidence,
            match_positions=positions[: self.config.max_evidence_per_pattern],
        )
        return bundle.to_dict()

    def _run_contradiction_analysis(
        self, text: str, metadata: dict[str, Any]
    ) -> dict[str, Any]:
        """Execute contradiction and temporal diagnostics across all dimensions."""

        if not self.contradiction_detector:
            raise PDETAnalysisException("Contradiction detector unavailable")

        plan_name = metadata.get("title", "Plan de Desarrollo")
        dimension_mapping = {
            CausalDimension.D1_INSUMOS: ContradictionPolicyDimension.DIAGNOSTICO,
            CausalDimension.D2_ACTIVIDADES: ContradictionPolicyDimension.ESTRATEGICO,
            CausalDimension.D3_PRODUCTOS: ContradictionPolicyDimension.PROGRAMATICO,
            CausalDimension.D4_RESULTADOS: ContradictionPolicyDimension.SEGUIMIENTO,
            CausalDimension.D5_IMPACTOS: ContradictionPolicyDimension.TERRITORIAL,
            CausalDimension.D6_CAUSALIDAD: ContradictionPolicyDimension.ESTRATEGICO,
        }

        domain_weights = {
            CausalDimension.D1_INSUMOS: 1.1,
            CausalDimension.D2_ACTIVIDADES: get_parameter_loader().get("saaaaaa.processing
.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1000_44", 1.0),
            CausalDimension.D3_PRODUCTOS: get_parameter_loader().get("saaaaaa.processing.p
olicy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1001_42", 1.0)5,
            CausalDimension.D4_RESULTADOS: 1.1,
            CausalDimension.D5_IMPACTOS: 1.15,
            CausalDimension.D6_CAUSALIDAD: 1.2,
        }

        reports: dict[str, Any] = {}
        temporal_assessments: dict[str, Any] = {}
        bayesian_scores: dict[str, float] = {}
        critical_links: dict[str, Any] = {}
        risk_assessment: dict[str, Any] = {}
        intervention_recommendations: dict[str, Any] = {}
        for dimension in CausalDimension:
            policy_dimension = dimension_mapping.get(dimension)
            try:
                report = self.contradiction_detector.detect(
                    text, plan_name=plan_name, dimension=policy_dimension
                )
            except Exception as exc:  # pragma: no cover - external deps
                raise PDETAnalysisException(
                    f"Contradiction detection failed for {dimension.name}: {exc}"
                ) from exc

            reports[dimension.value] = report

            try:
                statements = self.contradiction_detector._extract_policy_statements(  #
type: ignore[attr-defined]
                    text, policy_dimension
                )
            except Exception:  # pragma: no cover - best effort if detector lacks method
                statements = []

            is_consistent, conflicts = self.temporal_verifier.verify_temporal_consistency(
                statements
            )
            temporal_assessments[dimension.value] = {
                "is_consistent": is_consistent,
                "conflicts": conflicts,
            }

            coherence_metrics = report.get("coherence_metrics", {})
            coherence_score = float(coherence_metrics.get("coherence_score", get_parameter
_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get
("auto_param_L1043_77", 0.0)))
            observations = max(1, len(statements))
            posterior = self.confidence_calculator.calculate_posterior(
                evidence_strength=max(coherence_score, get_parameter_loader().get("saaaaaa
.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1046_55"
, 0.01)),
                observations=observations,
                domain_weight=domain_weights.get(dimension, get_parameter_loader().get("sa
aaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_L104
8_60", 1.0)),
            )
            bayesian_scores[dimension.value] = float(posterior)

            total_contradictions = int(report.get("total_contradictions", 0))
            if total_contradictions:
                keywords = []
                # Defensive: ensure contradictions is a list
                contradictions_list = ensure_list_return(report.get("contradictions", []))
                for contradiction in contradictions_list:
                    ctype = contradiction.get("contradiction_type")
                    if ctype:
                        keywords.append(ctype)

                severity = 1 - coherence_score if coherence_score else get_parameter_loade
r().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto
_param_L1062_71", 0.5)
                critical_links[dimension.value] = {
                    "criticality_score": round(min(get_parameter_loader().get("saaaaaa.pro
cessing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1064_51",
1.0), max(get_parameter_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicy
Processor.process").get("auto_param_L1064_60", 0.0), severity)), 4),
                    "text_analysis": {
                        "sentiment": "negative" if coherence_score < get_parameter_loader(
).get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_p
aram_L1066_69", 0.5) else "neutral",
                        "keywords": keywords,
                        "word_count": len(text.split()),
                    },
                }
                risk_assessment[dimension.value] = {
                    "overall_risk": "high" if total_contradictions > 3 else "medium",
                    "risk_factors": keywords,
                }
                intervention_recommendations[dimension.value] = report.get(
                    "recommendations", []
                )

            return {
            "reports": reports,
            "temporal_assessments": temporal_assessments,
            "bayesian_scores": bayesian_scores,
            "critical_diagnosis": {
                "critical_links": critical_links,
                "risk_assessment": risk_assessment,
                "intervention_recommendations": intervention_recommendations,
            },
            }

        def _calculate_quality_score(
            self,
            dimension_analysis: dict[str, Any],
            contradiction_bundle: dict[str, Any],
            performance_analysis: dict[str, Any],
        ) -> QualityScore:
            """Aggregate key indicators into a structured QualityScore dataclass."""

            bayesian_scores = contradiction_bundle.get("bayesian_scores", {})
            bayesian_values = list(bayesian_scores.values())
            overall_score = float(np.mean(bayesian_values)) if bayesian_values else get_parame
ter_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").
get("auto_param_L1100_80", 0.0)

            def _dimension_confidence(key: CausalDimension) -> float:
            return float(
                dimension_analysis.get(key.value, {}).get("dimension_confidence", get_para
meter_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process"
).get("auto_param_L1104_82", 0.0))
            )

            temporal_flags = contradiction_bundle.get("temporal_assessments", {})
            temporal_values = [
            get_parameter_loader().get("saaaaaa.processing.policy_processor.IndustrialPoli
cyProcessor.process").get("auto_param_L1109_12", 1.0) if assessment.get("is_consistent",
True) else get_parameter_loader().get("saaaaaa.processing.policy_processor.IndustrialPolic
yProcessor.process").get("auto_param_L1109_62", 0.0)
            for assessment in temporal_flags.values()
            ]
            temporal_consistency = (
            float(np.mean(temporal_values)) if temporal_values else get_parameter_loader()
.get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_pa
ram_L1113_68", 1.0)
            )

            reports = contradiction_bundle.get("reports", {})
            coherence_scores = [
            float(report.get("coherence_metrics", {}).get("coherence_score", get_parameter
_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get
("auto_param_L1118_77", 0.0)))
            for report in reports.values()
            ]
            causal_coherence = float(np.mean(coherence_scores)) if coherence_scores else get_p
arameter_loader().get("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.proce
ss").get("auto_param_L1121_85", 0.0)
```

```
        objective_alignment = float(
            reports.get(
                CausalDimension.D4_RESULTADOS.value,
                {},
            )
            .get("coherence_metrics", {})
            .get("objective_alignment", get_parameter_loader().get("saaaaaa.processing.pol
icy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1129_40", 0.0))
        )

        confidence_interval = (
            float(min(bayesian_values)) if bayesian_values else get_parameter_loader().get
("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_
L1133_64", 0.0),
            float(max(bayesian_values)) if bayesian_values else get_parameter_loader().get
("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_
L1134_64", 0.0),
        )

        evidence = {
            "bayesian_scores": bayesian_scores,
            "dimension_confidences": {
                key: value.get("dimension_confidence", get_parameter_loader().get("saaaaaa
.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_L1140_55"
, 0.0))
                for key, value in dimension_analysis.items()
            },
            "performance_metrics": performance_analysis.get("value_chain_metrics", {}),
        }

        return QualityScore(
            overall_score=overall_score,
            financial_feasibility=_dimension_confidence(CausalDimension.D1_INSUMOS),
            indicator_quality=_dimension_confidence(CausalDimension.D3_PRODUCTOS),
            responsibility_clarity=_dimension_confidence(CausalDimension.D2_ACTIVIDADES),
            temporal_consistency=temporal_consistency,
            pdet_alignment=objective_alignment,
            causal_coherence=causal_coherence,
            confidence_interval=confidence_interval,
            evidence=evidence,
        )

    def _extract_point_evidence(
        self, text: str, sentences: list[str], point_code: str
    ) -> dict[str, Any]:
        """Extract evidence for a specific policy point across all dimensions."""
        pattern = self.point_patterns.get(point_code)
        if not pattern:
            return {}

        # Find relevant sentences
        relevant_sentences = [s for s in sentences if pattern.search(s)]
        if not relevant_sentences:
            return {}

        # Search for dimensional evidence within relevant context
        evidence_by_dimension = {}
        for dimension, categories in self._pattern_registry.items():
            dimension_evidence = []

            for category, compiled_patterns in categories.items():
                matches, positions = self._match_patterns_in_sentences(
                    compiled_patterns, relevant_sentences
                )

                if matches:
                    confidence = self._compute_evidence_confidence(
                        matches, len(text), pattern_specificity=get_parameter_loader().get
("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_
L1183_64", 0.85)
                )

                if confidence >= self.config.confidence_threshold:
                    evidence_dict = self._construct_evidence_bundle(
                        dimension, category, matches, positions, confidence
                    )
                    dimension_evidence.append(evidence_dict)

            if dimension_evidence:
                evidence_by_dimension[dimension.value] = dimension_evidence

        return evidence_by_dimension

    def _analyze_causal_dimensions(
        self, text: str, sentences: list[str] | None = None
    ) -> dict[str, Any]:
        """
        Perform global analysis of causal dimensions across entire document.

        Args:
            text: Full document text
            sentences: Optional pre-segmented sentences. If not provided, will be
                     automatically extracted from text using the text processor.

        Returns:
            Dictionary containing dimension scores and confidence metrics

        Note:
            This function requires 'sentences' for optimal performance. If not provided,
            sentences will be extracted from text automatically, which may impact
performance.
        """
        # Defensive validation: ensure sentences parameter is provided
        if sentences is None:
            logger.warning(
                "_analyze_causal_dimensions called without 'sentences' parameter. "
                "Automatically extracting sentences from text. "
                "Expected signature: _analyze_causal_dimensions(self, text: str,
sentences: List[str])"
            )
            # Auto-extract sentences if not provided
            sentences = self.text_processor.segment_into_sentences(text)

        dimension_scores = {}

        for dimension, categories in self._pattern_registry.items():
            total_matches = 0
            category_results = {}

            for category, compiled_patterns in categories.items():
                matches = []
                for pattern in compiled_patterns:
                    for sentence in sentences:
                        matches.extend(pattern.findall(sentence))

                if matches:
                    confidence = self.scorer.compute_evidence_score(
                        matches, len(text), pattern_specificity=get_parameter_loader().get
("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_param_
L1239_64", 0.80)
                    )
                    category_results[category] = {
                        "match_count": len(matches),
                        "confidence": round(confidence, 4),
                    }
                    total_matches += len(matches)
            dimension_scores[dimension.value] = {
                "categories": category_results,
                "total_matches": total_matches,
                "dimension_confidence": round(
                    np.mean([c["confidence"] for c in category_results.values()])
                    if category_results
                    else get_parameter_loader().get("saaaaaa.processing.policy_processor.I
ndustrialPolicyProcessor.process").get("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 título identificado"

        # Entity extraction
        entity_match = re.search(
            r"(?i)(?:municipio|alcald[íi]a|gobernaci[óo]n|distrito)\s+(?:de\s+)?([A-
ZÁÉÍÓÚÑ][a-záéíóúñ\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*[-–—]\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 get_parameter_loader().g
et("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor.process").get("auto_para
m_L1299_66", 0.0)

    @calibrated_method("saaaaaa.processing.policy_processor.IndustrialPolicyProcessor._emp
ty_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": get_parameter_loader().get("saaaaaa.processing.policy_pr
ocessor.IndustrialPolicyProcessor._empty_result").get("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 .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("saaaaaa.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)

        # Stage 2: Protect structural elements
        if self.config.preserve_document_structure:
            text = self._protect_structure(text)

        # Stage 3: Whitespace normalization
        text = re.sub(r"[ \t]+", " ", text)
        text = re.sub(r"\n{3,}", "\n\n", text)
        # Stage 4: Remove control characters (except newlines/tabs)
        text = "".join(
            char for char in text
            if unicodedata.category(char)[0] != "C" or char in "\n\t"
        )

        # Stage 5: Restore protected elements
        if self.config.preserve_document_structure:
            text = self._restore_structure(text)

        return text.strip()

    @calibrated_method("saaaaaa.processing.policy_processor.AdvancedTextSanitizer._protect
_structure")
    def _protect_structure(self, text: str) -> str:
        """Mark structural elements for protection during sanitization."""
        protected = text

        # Protect headings (numbered or capitalized lines)
        heading_pattern = re.compile(
            r"^(?:[\d.]+\s+)?([A-ZÁÉÍÓÚÑ][A-ZÁÉÍÓÚÑa-záéíóúñ\s]{5,80})$",
            re.MULTILINE,
        )
        for match in reversed(list(heading_pattern.finditer(protected))):
            start, end = match.span()
            heading_text = match.group(0)
            protected = (
                protected[:start]
                + f"{self.protection_markers['heading'][0]}{heading_text}{self.protection_
markers['heading'][1]}"
                + protected[end:]
            )

        # Protect list items
        list_pattern = re.compile(r"^[\s]*[•\-\*\d]+[\.\)]\s+(.+)$", re.MULTILINE)
        for match in reversed(list(list_pattern.finditer(protected))):
            start, end = match.span()
            item_text = match.group(0)
            protected = (
                protected[:start]
                + f"{self.protection_markers['list_item'][0]}{item_text}{self.protection_m
arkers['list_item'][1]}"
                + protected[end:]
            )

        return protected

    @calibrated_method("saaaaaa.processing.policy_processor.AdvancedTextSanitizer._restore
_structure")
    def _restore_structure(self, text: str) -> str:
        """Remove protection markers after sanitization."""
        restored = text
        for _marker_type, (start_mark, end_mark) in self.protection_markers.items():
            restored = restored.replace(start_mark, "")
            restored = restored.replace(end_mark, "")
        return restored

# ============================================================================
# INTEGRATED FILE HANDLING WITH RESILIENCE
# ============================================================================

class ResilientFileHandler:
    """
    Production-grade file I/O with automatic encoding detection,
    retry logic, and comprehensive error classification.
    """

    ENCODINGS = ["utf-8", "utf-8-sig", "latin-1", "cp1252", "iso-8859-1"]
    @classmethod
    def read_text(cls, file_path: str | Path) -> str:
        """
        Read text file with automatic encoding detection and fallback cascade.

        Args:
            file_path: Path to input file

        Returns:
            Decoded text content

        Raises:
            IOError: If file cannot be read with any supported encoding
        """
        # Delegate to factory for I/O operation
        from .factory import read_text_file

        try:
            return read_text_file(file_path, encodings=list(cls.ENCODINGS))
        except Exception as e:
            raise OSError(f"Failed to read {file_path} with any supported encoding") from
e

    @classmethod
    def write_text(cls, content: str, file_path: str | Path) -> None:
        """Write text content with UTF-8 encoding and directory creation."""
        # Delegate to factory for I/O operation
        from .factory import write_text_file

        write_text_file(content, file_path)

# ============================================================================
# UNIFIED ORCHESTRATOR
# ============================================================================

class PolicyAnalysisPipeline:
    """
    End-to-end orchestrator for Colombian local development plan analysis
    implementing the complete DECALOGO causal framework evaluation workflow.

    DEPRECATION NOTE: The questionnaire_path parameter is deprecated.
    Modern pipelines use SPC (Smart Policy Chunks) ingestion which handles
    questionnaire integration separately.
    """

    def __init__(
        self,
        config: ProcessorConfig | None = None,
        questionnaire_path: Path | None = None,  # DEPRECATED: Kept for API compatibility
only
    ) -> 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.sanitizer = AdvancedTextSanitizer(self.config)

        # Initialize shared domain components
        self.ontology = MunicipalOntology()
        self.semantic_analyzer = SemanticAnalyzer(self.ontology)
        self.performance_analyzer = PerformanceAnalyzer(self.ontology)
        self.temporal_verifier = TemporalLogicVerifier()
        self.confidence_calculator = BayesianConfidenceCalculator()
        self.contradiction_detector = PolicyContradictionDetector()
        self.municipal_analyzer = MunicipalAnalyzer()

        self.processor = IndustrialPolicyProcessor(
            self.config,
            questionnaire_path,
            ontology=self.ontology,
            semantic_analyzer=self.semantic_analyzer,
            performance_analyzer=self.performance_analyzer,
            contradiction_detector=self.contradiction_detector,
            temporal_verifier=self.temporal_verifier,
            confidence_calculator=self.confidence_calculator,
            municipal_analyzer=self.municipal_analyzer,
        )
        self.file_handler = ResilientFileHandler()

    def analyze_file(
        self,
        input_path: str | Path,
        output_path: str | Path | None = None,
    ) -> dict[str, Any]:
        """
        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 = DocumentProcessor.load_pdf(str(input_path))
        elif suffix in {".docx", ".doc"}:
            raw_text = DocumentProcessor.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("saaaaaa.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
        """
        sanitized_text = self.sanitizer.sanitize(raw_text)
        return self.processor.process(sanitized_text)

# ============================================================================
# FACTORY FUNCTIONS FOR BACKWARD COMPATIBILITY
# ============================================================================

def create_policy_processor(
    preserve_structure: bool = True,
    enable_semantic_tagging: bool = True,
    confidence_threshold: float = get_parameter_loader().get("saaaaaa.processing.policy_pr
ocessor.PolicyAnalysisPipeline.analyze_text").get("auto_param_L1595_34", 0.65),
    **kwargs: Any,
) -> PolicyAnalysisPipeline:
    """
    Factory function for creating policy analysis pipeline with legacy support.

    Args:
        preserve_structure: Enable document structure preservation
        enable_semantic_tagging: Enable semantic element tagging
        confidence_threshold: Minimum confidence threshold for evidence
        **kwargs: Additional configuration parameters

    Returns:
        Configured PolicyAnalysisPipeline instance
    """
    config = ProcessorConfig(
        preserve_document_structure=preserve_structure,
        enable_semantic_tagging=enable_semantic_tagging,
        confidence_threshold=confidence_threshold,
        **kwargs,
    )
    return PolicyAnalysisPipeline(config=config)

# ============================================================================
# COMMAND-LINE INTERFACE
# ============================================================================

def main() -> None:
    """Command-line interface for policy plan analysis."""
    import argparse

    parser = argparse.ArgumentParser(
        description="Industrial-Grade Policy Plan Processor for Colombian Local
Development Plans"
    )
    parser.add_argument("input_file", type=str, help="Input policy document path")
    parser.add_argument(
        "-o", "--output", type=str, help="Output JSON file path", default=None
    )
    parser.add_argument(
        "-t",
        "--threshold",
        type=float,
        default=get_parameter_loader().get("saaaaaa.processing.policy_processor.PolicyAnal
ysisPipeline.analyze_text").get("auto_param_L1637_16", 0.65),
        help="Confidence threshold (0-1)",
    )
    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/saaaaaa/processing/semantic_chunking_policy.py =====
"""
INTERNAL SPC COMPONENT

⚠  USAGE RESTRICTION ⚠
============================================================================
This module implements SOTA semantic chunking 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:
    scripts/smart_policy_chunks_canonic_phase_one.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:
    - smart_policy_chunks_canonic_phase_one.py
    - Unit tests for SPC components

Scientific Foundation:
- Semantic: BGE-M3 (2024, SOTA multilingual dense retrieval)
- 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 saaaaaa.core.dependency_lockdown import get_dependency_lockdown
from transformers import AutoModel, AutoTokenizer
from saaaaaa import get_parameter_loader
from saaaaaa.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
PLAN_SECTION_WEIGHT_FACTOR: float = 1.25  # Investment plans anchor execution feasibility
DIAGNOSTIC_SECTION_WEIGHT_FACTOR: float = 0.92  # Diagnostics contextualize but do not
commit resources
RENYI_ALPHA_ORDER: float = 1.45  # Van Erven & Harremoës (2014) Optimum between KL and
Rényi regimes
RENYI_ALERT_THRESHOLD: float = 0.24  # Empirically tuned on 2021-2024 Colombian PDM corpus
RENYI_CURVATURE_GAIN: float = 0.85  # Amplifies curvature impact without destabilizing
```

```
evidence
RENYI_FLUX_TEMPERATURE: float = 0.65  # Controls saturation of Renyi coherence flux
RENYI_STABILITY_EPSILON: float = 1e-9  # Numerical guard-rail for degenerative posteriors

# ========================
# DOMAIN ONTOLOGY
# ========================

class CausalDimension(Enum):
    """Marco Lógico standard (DNP Colombia)"""
    INSUMOS = "insumos"  # Recursos, capacidad institucional
    ACTIVIDADES = "actividades"  # Acciones, procesos, cronogramas
    PRODUCTOS = "productos"  # Entregables inmediatos
    RESULTADOS = "resultados"  # Efectos mediano plazo
    IMPACTOS = "impactos"  # Transformación estructural largo plazo
    SUPUESTOS = "supuestos"  # Condiciones habilitantes

class PDMSection(Enum):
    """
    Enumerates the typical sections of a Colombian Municipal Development Plan (PDM),
    as defined by Ley 152/1994. Each member represents a key structural component
    of the PDM document, facilitating semantic analysis and policy structure recognition.
    """
    DIAGNOSTICO = "diagnostico"
    VISION_ESTRATEGICA = "vision_estrategica"
    PLAN_PLURIANUAL = "plan_plurianual"
    PLAN_INVERSIONES = "plan_inversiones"
    MARCO_FISCAL = "marco_fiscal"
    SEGUIMIENTO = "seguimiento_evaluacion"

@dataclass(frozen=True, slots=True)
class SemanticConfig:
    """Configuración calibrada para análisis de políticas públicas"""
    # BGE-M3: Best multilingual embedding (Jan 2024, beats E5)
    embedding_model: str = "BAAI/bge-m3"
    chunk_size: int = 768  # Optimal for policy paragraphs (empirical)
    chunk_overlap: int = 128  # Preserve cross-boundary context
    similarity_threshold: float = 0.82  # Calibrated on PDM corpus
    min_evidence_chunks: int = 3  # Statistical significance floor
    bayesian_prior_strength: float = 0.5  # Conservative uncertainty
    device: Literal["cpu", "cuda"] | None = None
    batch_size: int = 32
    fp16: bool = True  # Memory optimization

# ========================
# SEMANTIC PROCESSOR (SOTA)
# ========================

class SemanticProcessor:
    """
    State-of-the-art semantic processing with:
    - 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("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor._laz
y_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("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor.chun
k_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("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor._det
ect_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)(diagnóstico|caracterización|situación actual)",
            PDMSection.VISION_ESTRATEGICA: r"(?i)(visión|misión|objetivos estratégicos)",
            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|evaluación|indicadores)"
        }
        # Split by major headers (numbered or capitalized)
        parts = re.split(r'\n(?=[0-9]+\.|[A-ZÑÁÉÍÓÚ]{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("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor._det
ect_table")
    def _detect_table(self, text: str) -> bool:
        """Detect if chunk contains tabular data"""
        # Multiple tabs or pipes suggest table structure
        return (text.count('\t') > 3 or
                text.count('|') > 3 or
                bool(re.search(r'\d+\s+\d+\s+\d+', text)))

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor._det
ect_numerical_data")
    def _detect_numerical_data(self, text: str) -> bool:
        """Detect if chunk contains significant numerical/financial data"""
        # Look for currency, percentages, large numbers
        patterns = [
            r'\$\s*\d+(?:[\.,]\d+)*',  # Currency
            r'\d+(?:[\.,]\d+)*\s*%',  # Percentages
            r'\d{1,3}(?:[.,]\d{3})+',  # Large numbers with separators
        ]
        return any(re.search(p, text) for p in patterns)

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor._emb
ed_batch")
    def _embed_batch(self, texts: list[str]) -> list[NDArray[np.floating[Any]]]:
        """Batch embedding with BGE-M3"""
        self._lazy_load()
        embeddings = []
        for i in range(0, len(texts), self.config.batch_size):
            batch = texts[i:i + self.config.batch_size]
            # Tokenize batch
            encoded = self._tokenizer(
                batch,
                padding=True,
                truncation=True,
                max_length=self.config.chunk_size,
                return_tensors="pt"
            ).to(self._model.device)
            # Generate embeddings (mean pooling)
            with torch.no_grad():
                outputs = self._model(**encoded)
            # Mean pooling over sequence
            attention_mask = encoded["attention_mask"]
            token_embeddings = outputs.last_hidden_state
            input_mask_expanded =
attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
            sum_embeddings = torch.sum(token_embeddings * input_mask_expanded, 1)
            sum_mask = torch.clamp(input_mask_expanded.sum(1), min=1e-9)
            batch_embeddings = (sum_embeddings / sum_mask).cpu().numpy()
            embeddings.extend([emb.astype(np.float32) for emb in batch_embeddings])
        return embeddings

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticProcessor.embe
d_single")
    def embed_single(self, text: str) -> NDArray[np.floating[Any]]:
        """Single text embedding"""
        return self._embed_batch([text])[0]

# ========================
# MATHEMATICAL ENHANCER (RIGOROUS)
# ========================

class BayesianEvidenceIntegrator:
    """
    Information-theoretic Bayesian evidence accumulation:
    - Dirichlet-Multinomial for multi-hypothesis tracking
    - KL divergence for belief update quantification
    - Entropy-based confidence calibration
    - No simplifications or heuristics
    """

    def __init__(self, prior_concentration: float = 0.5) -> None:
        """
        Args:
            prior_concentration: Dirichlet concentration (α).
                Lower = more uncertain prior (conservative)
        """
        if prior_concentration <= 0:
            raise ValueError(
                "Invalid prior_concentration: Dirichlet concentration parameter (α) must
be strictly positive. "
                "Typical values are in the range get_parameter_loader().get("saaaaaa.proce
ssing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__").get("auto_param_L318_
49", 0.1)–get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.Bayesian
EvidenceIntegrator.__init__").get("auto_param_L318_53", 1.0) for conservative priors. "
                "Lower values (e.g., get_parameter_loader().get("saaaaaa.processing.semant
ic_chunking_policy.BayesianEvidenceIntegrator.__init__").get("auto_param_L319_37", 0.1))
indicate greater prior uncertainty; higher values (e.g., get_parameter_loader().get("saaaa
aa.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__").get("auto_par
am_L319_99", 1.0)) indicate stronger prior beliefs. "
                f"Received: {prior_concentration}"
            )
        self.prior_alpha = float(prior_concentration)

    def integrate_evidence(
        self,
        similarities: NDArray[np.float64],
        chunk_metadata: list[dict[str, Any]]
    ) -> dict[str, float]:
        """
        Bayesian evidence integration with information-theoretic rigor:
        1. Map similarities to likelihood space via monotonic transform
        2. Weight evidence by chunk reliability (position, structure, content type)
        3. Update Dirichlet posterior
        4. Compute information gain (KL divergence from prior)
        5. Calculate calibrated confidence with epistemic uncertainty
        """
        if len(similarities) == 0:
            return self._null_evidence()
        # 1. Transform similarities to probability space
        # Using sigmoid with learned temperature for calibration
        sims = np.asarray(similarities, dtype=np.float64)
        probs = self._similarity_to_probability(sims)
        # 2. Compute reliability weights from metadata
        weights = self._compute_reliability_weights(chunk_metadata)
        # 3. Aggregate weighted evidence
        # Dirichlet posterior parameters: α_post = α_prior + weighted_counts
        positive_evidence = np.sum(weights * probs)
        negative_evidence = np.sum(weights * (get_parameter_loader().get("saaaaaa.processi
ng.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__").get("auto_param_L348_46"
, 1.0) - probs))
        alpha_pos = self.prior_alpha + positive_evidence
        alpha_neg = self.prior_alpha + negative_evidence
        alpha_total = alpha_pos + alpha_neg
        # 4. Posterior statistics
        posterior_mean = alpha_pos / alpha_total
        posterior_variance = (alpha_pos * alpha_neg) / (
            alpha_total**2 * (alpha_total + 1)
        )
        # 5. Information gain (KL divergence from prior to posterior)
        prior_dist = np.array([self.prior_alpha, self.prior_alpha])
        prior_dist = prior_dist / prior_dist.sum()
        posterior_dist = np.array([alpha_pos, alpha_neg])
        posterior_dist = posterior_dist / posterior_dist.sum()
        kl_divergence = float(np.sum(rel_entr(posterior_dist, prior_dist)))
        # 6. Entropy-based calibrated confidence
        posterior_entropy = stats.beta.entropy(alpha_pos, alpha_neg)
        max_entropy = stats.beta.entropy(1, 1)  # Maximum uncertainty
        confidence = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_poli
cy.BayesianEvidenceIntegrator.__init__").get("auto_param_L366_21", 1.0) -
(posterior_entropy / max_entropy)
        return {
            "posterior_mean": float(np.clip(posterior_mean, get_parameter_loader().get("sa
aaaaa.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__").get("auto_
param_L368_60", 0.0), get_parameter_loader().get("saaaaaa.processing.semantic_chunking_pol
icy.BayesianEvidenceIntegrator.__init__").get("auto_param_L368_65", 1.0))),
            "posterior_std": float(np.sqrt(posterior_variance)),
            "information_gain": float(kl_divergence),
            "confidence": float(confidence),
            "evidence_strength": float(
                positive_evidence / (alpha_total - 2 * self.prior_alpha)
                if abs(alpha_total - 2 * self.prior_alpha) > 1e-8 else get_parameter_loade
r().get("saaaaaa.processing.semantic_chunking_policy.BayesianEvidenceIntegrator.__init__")
.get("auto_param_L374_71", 0.0)
            ),
            "n_chunks": len(similarities)
        }

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.BayesianEvidenceIntegr
ator._similarity_to_probability")
    def _similarity_to_probability(self, sims: NDArray[np.float64]) ->
NDArray[np.float64]:
        """
        Calibrated transform from cosine similarity [-1,1] to probability [0,1]
        Using sigmoid with empirically derived temperature
        """
        # Shift to [0,2], scale to reasonable range
        x = (sims + get_parameter_loader().get("saaaaaa.processing.semantic_chunking_polic
y.BayesianEvidenceIntegrator._similarity_to_probability").get("auto_param_L386_20", 1.0))
* 2.0
```

```
        # Sigmoid with temperature=2.0 (calibrated on policy corpus)
        return get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.Bay
esianEvidenceIntegrator._similarity_to_probability").get("auto_param_L388_15", 1.0) / (get
_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.BayesianEvidenceInteg
rator._similarity_to_probability").get("auto_param_L388_22", 1.0) + np.exp(-x / 2.0))

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.BayesianEvidenceIntegr
ator._compute_reliability_weights")
    def _compute_reliability_weights(self, metadata: list[dict[str, Any]]) ->
NDArray[np.float64]:
        """
        Evidence reliability based on:
        - Position in document (early sections more diagnostic)
        - Content type (tables/numbers more reliable for quantitative claims)
        - Section type (plan sections more reliable than diagnostics)
        """
        n = len(metadata)
        weights = np.ones(n, dtype=np.float64)
        for i, meta in enumerate(metadata):
            # Position weight (early = more reliable)
            pos_weight = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_
policy.BayesianEvidenceIntegrator._compute_reliability_weights").get("auto_param_L402_25",
 1.0) - (meta["position"] / max(1, n)) * POSITION_WEIGHT_SCALE
            # Content type weight
            content_weight = get_parameter_loader().get("saaaaaa.processing.semantic_chunk
ing_policy.BayesianEvidenceIntegrator._compute_reliability_weights").get("content_weight",
 1.0) # Refactored
            if meta.get("has_table", False):
                content_weight *= TABLE_WEIGHT_FACTOR
            if meta.get("has_numerical", False):
                content_weight *= NUMERICAL_WEIGHT_FACTOR
            # Section type weight (plan sections > diagnostic)
            section_type = meta.get("section_type")
            if section_type in [PDMSection.PLAN_PLURIANUAL, PDMSection.PLAN_INVERSIONES]:
                content_weight *= PLAN_SECTION_WEIGHT_FACTOR
            elif section_type == PDMSection.DIAGNOSTICO:
                content_weight *= DIAGNOSTIC_SECTION_WEIGHT_FACTOR
            weights[i] = pos_weight * content_weight
        # Normalize to sum to n (preserve total evidence mass)
        return weights * (n / weights.sum())

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.BayesianEvidenceIntegr
ator._null_evidence")
    def _null_evidence(self) -> dict[str, float]:
        """Return prior state (no evidence)"""
        prior_mean = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_poli
cy.BayesianEvidenceIntegrator._null_evidence").get("prior_mean", 0.5) # Refactored
        prior_var = self.prior_alpha / \
            ((2 * self.prior_alpha)**2 * (2 * self.prior_alpha + 1))
        return {
            "posterior_mean": prior_mean,
            "posterior_std": float(np.sqrt(prior_var)),
            "information_gain": get_parameter_loader().get("saaaaaa.processing.semantic_ch
unking_policy.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L428_32", 0.0),
            "confidence": get_parameter_loader().get("saaaaaa.processing.semantic_chunking
_policy.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L429_26", 0.0),
            "evidence_strength": get_parameter_loader().get("saaaaaa.processing.semantic_c
hunking_policy.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L430_33", 0.0),
            "n_chunks": 0
        }

    def causal_strength(
        self,
        cause_emb: NDArray[np.floating[Any]],
        effect_emb: NDArray[np.floating[Any]],
        context_emb: NDArray[np.floating[Any]]
    ) -> float:
        """
        Causal strength via conditional independence approximation:
        strength = sim(cause, effect) * [1 - |sim(cause,ctx) - sim(effect,ctx)|]
        Intuition: Strong causal link if cause-effect similar AND
        both relate similarly to context (conditional independence test proxy)
        """
        sim_ce = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.B
ayesianEvidenceIntegrator._null_evidence").get("auto_param_L446_17", 1.0) -
cosine(cause_emb, effect_emb)
        sim_c_ctx = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_polic
y.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L447_20", 1.0) -
cosine(cause_emb, context_emb)
        sim_e_ctx = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_polic
y.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L448_20", 1.0) -
cosine(effect_emb, context_emb)
        # Conditional independence proxy
        cond_indep = get_parameter_loader().get("saaaaaa.processing.semantic_chunking_poli
cy.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L450_21", 1.0) -
abs(sim_c_ctx - sim_e_ctx)
        # Combined strength (normalized to [0,1])
        strength = ((sim_ce + 1) / 2) * cond_indep
        return float(np.clip(strength, get_parameter_loader().get("saaaaaa.processing.sema
ntic_chunking_policy.BayesianEvidenceIntegrator._null_evidence").get("auto_param_L453_39",
 0.0), get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.BayesianEvi
denceIntegrator._null_evidence").get("auto_param_L453_44", 1.0)))

# ========================
# POLICY ANALYZER (INTEGRATED)
# ========================

class PolicyDocumentAnalyzer:
    """
    Colombian Municipal Development Plan Analyzer:
    - BGE-M3 semantic processing
    - Policy-aware chunking (respects PDM structure)
    - Bayesian evidence integration with information theory
    - Causal dimension analysis per Marco Lógico
    """

    def __init__(self, config: SemanticConfig | None = None) -> None:
        self.config = config or SemanticConfig()
        self.semantic = SemanticProcessor(self.config)
        self.bayesian = BayesianEvidenceIntegrator(
            prior_concentration=self.config.bayesian_prior_strength
        )
        # Initialize dimension embeddings
        self.dimension_embeddings = self._init_dimension_embeddings()

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.PolicyDocumentAnalyzer
._init_dimension_embeddings")
    def _init_dimension_embeddings(self) -> dict[CausalDimension,
NDArray[np.floating[Any]]]:
        """
        Canonical embeddings for Marco Lógico dimensions
        Using Colombian policy-specific terminology
        """
        descriptions = {
            CausalDimension.INSUMOS: (
                "recursos humanos financieros técnicos capacidad institucional "
                "presupuesto asignado infraestructura disponible personal capacitado"
            ),
            CausalDimension.ACTIVIDADES: (
                "actividades programadas acciones ejecutadas procesos implementados "
                "cronograma cumplido capacitaciones realizadas gestiones adelantadas"
            ),
            CausalDimension.PRODUCTOS: (
                "productos entregables resultados inmediatos bienes servicios generados "
                "documentos producidos obras construidas beneficiarios atendidos"
            ),
            CausalDimension.RESULTADOS: (
                "resultados efectos mediano plazo cambios comportamiento acceso mejorado "
                "capacidades fortalecidas servicios prestados metas alcanzadas"
            ),
            CausalDimension.IMPACTOS: (
                "impactos transformación estructural efectos largo plazo desarrollo
sostenible "
                "bienestar poblacional reducción pobreza equidad territorial"
            ),
            CausalDimension.SUPUESTOS: (
                "supuestos condiciones habilitantes riesgos externos factores contextuales
 "
                "viabilidad política sostenibilidad financiera apropiación comunitaria"
            )
        }
        return {
            dim: self.semantic.embed_single(desc)
            for dim, desc in descriptions.items()
        }

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.PolicyDocumentAnalyzer
.analyze")
    def analyze(self, text: str) -> dict[str, Any]:
        """
        Full pipeline: chunking → embedding → dimension analysis → evidence integration
        """
        # 1. Policy-aware chunking
        chunks = self.semantic.chunk_text(text, preserve_structure=True)
        logger.info(f"Processing {len(chunks)} chunks")
        # 2. Analyze each causal dimension
        dimension_results = {}
        for dim, dim_emb in self.dimension_embeddings.items():
            similarities = np.array([
                get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.Po
licyDocumentAnalyzer.analyze").get("auto_param_L526_16", 1.0) - cosine(chunk["embedding"],
 dim_emb)
                for chunk in chunks
            ])
            # Filter by threshold
            relevant_mask = similarities >= self.config.similarity_threshold
            relevant_sims = similarities[relevant_mask]
            relevant_chunks = [c for c, m in zip(chunks, relevant_mask, strict=False) if
m]
            # Bayesian integration
            if len(relevant_sims) >= self.config.min_evidence_chunks:
                evidence = self.bayesian.integrate_evidence(
                    relevant_sims,
                    relevant_chunks
                )
            else:
                evidence = self.bayesian._null_evidence()
            dimension_results[dim.value] = {
                "total_chunks": int(np.sum(relevant_mask)),
                "mean_similarity": float(np.mean(similarities)),
                "max_similarity": float(np.max(similarities)),
                **evidence
            }
        # 3. Extract key findings (top chunks per dimension)
        key_excerpts = self._extract_key_excerpts(chunks, dimension_results)
        return {
            "summary": {
                "total_chunks": len(chunks),
                "sections_detected": len({c["section_type"] for c in chunks}),
                "has_tables": sum(1 for c in chunks if c["has_table"]),
                "has_numerical": sum(1 for c in chunks if c["has_numerical"])
            },
            "causal_dimensions": dimension_results,
            "key_excerpts": key_excerpts
        }

    def _extract_key_excerpts(
        self,
        chunks: list[dict[str, Any]],
        dimension_results: dict[str, dict[str, Any]]
    ) -> dict[str, list[str]]:
        """Extract most relevant text excerpts per dimension"""
        _ = dimension_results  # parameter kept for future compatibility
        excerpts = {}
        for dim, dim_emb in self.dimension_embeddings.items():
            # Rank chunks by similarity
            sims = [
                (i, get_parameter_loader().get("saaaaaa.processing.semantic_chunking_polic
y.PolicyDocumentAnalyzer.analyze").get("auto_param_L571_20", 1.0) -
cosine(chunk["embedding"], dim_emb))
                for i, chunk in enumerate(chunks)
            ]
            sims.sort(key=lambda x: x[1], reverse=True)
            # Top 3 excerpts
            top_chunks = [chunks[i] for i, _ in sims[:3]]
            excerpts[dim.value] = [
                _get_chunk_content(c)[:300]
                + ("..." if len(_get_chunk_content(c)) > 300 else "")
                for c in top_chunks
            ]
        return excerpts

# ========================
# PRODUCER CLASS - Registry Exposure
# ========================

class SemanticChunkingProducer:
    """
    Producer wrapper for semantic chunking and policy analysis with registry exposure

    Provides public API methods for orchestrator integration without exposing
    internal implementation details or summarization logic.

    Version: get_parameter_loader().get("saaaaaa.processing.semantic_chunking_policy.Polic
yDocumentAnalyzer.analyze").get("auto_param_L595_13", 1.0).0
    Producer Type: Semantic Analysis / Chunking
    """

    def __init__(self, config: SemanticConfig | None = None) -> None:
        """Initialize producer with optional configuration"""
        self.config = config or SemanticConfig()
        self.semantic = SemanticProcessor(self.config)
        self.bayesian = BayesianEvidenceIntegrator(
            prior_concentration=self.config.bayesian_prior_strength
        )
        self.analyzer = PolicyDocumentAnalyzer(self.config)
        logger.info("SemanticChunkingProducer initialized")

    # ========================================================================
    # CHUNKING API
    # ========================================================================

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.chunk_document")
    def chunk_document(self, text: str, preserve_structure: bool = True) -> list[dict[str,
 Any]]:
        """Chunk document into semantic units with embeddings"""
        return self.semantic.chunk_text(text, preserve_structure)

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_chunk_count")
    def get_chunk_count(self, chunks: list[dict[str, Any]]) -> int:
        """Get number of chunks"""
        return len(chunks)

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

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.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("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.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("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.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("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.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("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.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, {})

    def get_dimension_score(
        self,
        analysis: dict[str, Any],
        dimension: CausalDimension
    ) -> float:
        """Extract dimension evidence strength score"""
        dim_result = self.get_dimension_analysis(analysis, dimension)
        return dim_result.get("evidence_strength", get_parameter_loader().get("saaaaaa.pro
cessing.semantic_chunking_policy.SemanticChunkingProducer.analyze_document").get("auto_par
am_L683_51", 0.0))

    def get_dimension_confidence(
        self,
        analysis: dict[str, Any],
        dimension: CausalDimension
    ) -> float:
        """Extract dimension confidence score"""
        dim_result = self.get_dimension_analysis(analysis, dimension)
        return dim_result.get("confidence", get_parameter_loader().get("saaaaaa.processing
.semantic_chunking_policy.SemanticChunkingProducer.analyze_document").get("auto_param_L692
_44", 0.0))

    def get_dimension_excerpts(
        self,
        analysis: dict[str, Any],
        dimension: CausalDimension
    ) -> list[str]:
        """Extract key excerpts for dimension"""
        return analysis.get("key_excerpts", {}).get(dimension.value, [])

    # ========================================================================
    # BAYESIAN EVIDENCE API
    # ========================================================================

    def integrate_evidence(
        self,
        similarities: NDArray[np.float64],
        chunk_metadata: list[dict[str, Any]]
    ) -> dict[str, float]:
        """Perform Bayesian evidence integration"""
        return self.bayesian.integrate_evidence(similarities, chunk_metadata)

    def calculate_causal_strength(
        self,
        cause_emb: NDArray[np.floating[Any]],
        effect_emb: NDArray[np.floating[Any]],
        context_emb: NDArray[np.floating[Any]]
    ) -> float:
        """Calculate causal strength between embeddings"""
        return self.bayesian.causal_strength(cause_emb, effect_emb, context_emb)

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_posterior_mean")
    def get_posterior_mean(self, evidence: dict[str, float]) -> float:
        """Extract posterior mean from evidence integration"""
        return evidence.get("posterior_mean", get_parameter_loader().get("saaaaaa.processi
ng.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_mean").get("auto_param_
L726_46", 0.0))

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_posterior_std")
    def get_posterior_std(self, evidence: dict[str, float]) -> float:
        """Extract posterior standard deviation"""
        return evidence.get("posterior_std", get_parameter_loader().get("saaaaaa.processin
g.semantic_chunking_policy.SemanticChunkingProducer.get_posterior_std").get("auto_param_L7
31_45", 0.0))

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_information_gain")
    def get_information_gain(self, evidence: dict[str, float]) -> float:
        """Extract information gain (KL divergence)"""
        return evidence.get("information_gain", get_parameter_loader().get("saaaaaa.proces
sing.semantic_chunking_policy.SemanticChunkingProducer.get_information_gain").get("auto_pa
ram_L736_48", 0.0))
```

```
    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_confidence")
    def get_confidence(self, evidence: dict[str, float]) -> float:
        """Extract confidence score"""
        return evidence.get("confidence", get_parameter_loader().get("saaaaaa.processing.s
emantic_chunking_policy.SemanticChunkingProducer.get_confidence").get("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 = get_parameter_loader().get("saaaaaa.processing.semantic_chunk
ing_policy.SemanticChunkingProducer.get_confidence").get("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("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.list_dimensions")
    def list_dimensions(self) -> list[CausalDimension]:
        """List all causal dimensions"""
        return list(CausalDimension)

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_dimension_description")
    def get_dimension_description(self, dimension: CausalDimension) -> str:
        """Get description for dimension"""
        descriptions = {
            CausalDimension.INSUMOS: "Recursos, capacidad institucional",
            CausalDimension.ACTIVIDADES: "Acciones, procesos, cronogramas",
            CausalDimension.PRODUCTOS: "Entregables inmediatos",
            CausalDimension.RESULTADOS: "Efectos mediano plazo",
            CausalDimension.IMPACTOS: "Transformación estructural largo plazo",
            CausalDimension.SUPUESTOS: "Condiciones habilitantes"
        }
        return descriptions.get(dimension, "")

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.get_config")
    def get_config(self) -> SemanticConfig:
        """Get current configuration"""
        return self.config

    @calibrated_method("saaaaaa.processing.semantic_chunking_policy.SemanticChunkingProduc
er.set_config")
    def set_config(self, config: SemanticConfig) -> None:
        """Update configuration (requires reinitialization)"""
        self.config = config
        self.semantic = SemanticProcessor(self.config)
        self.bayesian = BayesianEvidenceIntegrator(
            prior_concentration=self.config.bayesian_prior_strength
        )
        self.analyzer = PolicyDocumentAnalyzer(self.config)

# ========================
# CLI INTERFACE
# ========================

def main() -> None:
    """Example usage"""
    sample_pdm = """
PLAN DE DESARROLLO MUNICIPAL 2024-2027
MUNICIPIO DE EJEMPLO, COLOMBIA

1. DIAGNÓSTICO TERRITORIAL
El municipio cuenta con 45,000 habitantes, de los cuales 60% reside en zona rural.
La tasa de pobreza multidimensional es 42.3%, superior al promedio departamental.

2. VISIÓN ESTRATÉGICA
Para 2027, el municipio será reconocido por su desarrollo sostenible e inclusivo.

3. PLAN PLURIANUAL DE INVERSIONES
Se destinarán $12,500 millones al sector educación, con meta de construir
3 instituciones educativas y capacitar 250 docentes en pedagogías innovadoras.

4. SEGUIMIENTO Y EVALUACIÓN
Se implementará sistema de indicadores alineado con ODS, con mediciones semestrales.
"""
    config = SemanticConfig(
        chunk_size=512,
        chunk_overlap=100,
        similarity_threshold = get_parameter_loader().get("saaaaaa.processing.semantic_chu
nking_policy.SemanticChunkingProducer.set_config").get("similarity_threshold", 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/saaaaaa/processing/spc_ingestion/__init__.py =====
"""
SPC (Smart Policy Chunks) Ingestion - Canonical Phase-One
=============================================================

This module provides the canonical phase-one ingestion pipeline for processing
development plans into smart policy chunks with comprehensive analysis.

Main exports:
- CPPIngestionPipeline: Primary ingestion pipeline (for compatibility)
```

```python
- StrategicChunkingSystem: Core chunking system from smart_policy_chunks_canonic_phase_one

The pipeline performs:
1. Document preprocessing and structural analysis
2. Topic modeling and knowledge graph construction
3. Causal chain extraction
4. Temporal, argumentative, and discourse analysis
5. Smart chunk creation with inter-chunk relationships
6. Quality validation and strategic ranking
"""

import importlib.util
import logging
import unicodedata  # For NFC normalization
from pathlib import Path
from typing import Any

from saaaaaa.config.paths import QUESTIONNAIRE_FILE
from saaaaaa.processing.cpp_ingestion.models import CanonPolicyPackage
from saaaaaa.processing.spc_ingestion.converter import SmartChunkConverter
from saaaaaa import get_parameter_loader
from saaaaaa.processing.spc_ingestion.quality_gates import SPCQualityGates

logger = logging.getLogger(__name__)

# Load smart_policy_chunks_canonic_phase_one without sys.path manipulation
_root = Path(__file__).parent.parent.parent.parent.parent
_module_path = _root / "scripts" / "smart_policy_chunks_canonic_phase_one.py"

spec = importlib.util.spec_from_file_location(
    "smart_policy_chunks_canonic_phase_one",
    _module_path
)
if spec and spec.loader:
    _module = importlib.util.module_from_spec(spec)
    spec.loader.exec_module(_module)
    StrategicChunkingSystem = _module.StrategicChunkingSystem
else:
    raise ImportError(f"Cannot load smart_policy_chunks_canonic_phase_one from
{_module_path}")


class CPPIngestionPipeline:
    """
    SPC ingestion pipeline with orchestrator-compatible output.

    This class provides the canonical phase-one ingestion pipeline:
    1. Processes documents through StrategicChunkingSystem (15-phase analysis)
    2. Converts SmartPolicyChunk output to CanonPolicyPackage format
    3. Returns orchestrator-ready CanonPolicyPackage

    The pipeline ensures 100% alignment between SPC phase-one output and
    what the orchestrator expects to receive.

    Questionnaire Input Contract (SIN_CARRETA compliance):
    -------------------------------------------------------
    - questionnaire_path is an EXPLICIT input (defaults to canonical path)
    - Must be deterministic, auditable, and manifest-tracked
    - No hidden filesystem dependencies
    """

    def __init__(
        self,
        questionnaire_path: Path | None = None,
        enable_runtime_validation: bool = True,
    ) -> None:
        """
        Initialize the SPC ingestion pipeline with converter.
        Args:
            questionnaire_path: Optional path to questionnaire file.
                              If None, uses canonical path from
saaaaaa.config.paths.QUESTIONNAIRE_FILE
            enable_runtime_validation: Enable WiringValidator for runtime contract
checking
        """
        logger.info("Initializing CPPIngestionPipeline with StrategicChunkingSystem")

        # Store questionnaire path for manifest traceability
        if questionnaire_path is None:
            questionnaire_path = QUESTIONNAIRE_FILE

        self.questionnaire_path = questionnaire_path
        logger.info(f"Questionnaire path: {self.questionnaire_path}")

        self.chunking_system = StrategicChunkingSystem()
        self.converter = SmartChunkConverter()
        self.quality_gates = SPCQualityGates()

        # Initialize WiringValidator for runtime contract validation
        self.enable_runtime_validation = enable_runtime_validation
        if enable_runtime_validation:
            try:
                from saaaaaa.core.wiring.validation import WiringValidator
                self.wiring_validator = WiringValidator()
                logger.info("WiringValidator enabled for runtime contract checking")
            except ImportError:
                logger.warning(
                    "WiringValidator not available. Runtime validation disabled."
                )
                self.wiring_validator = None
        else:
            self.wiring_validator = None

        logger.info("Pipeline initialized successfully")

    def _load_document_text(self, document_path: Path) -> str:
        """
        Load document text from PDF, TXT, or MD files.

        Args:
            document_path: Path to document file

        Returns:
            Extracted text content

        Raises:
            ValueError: If file type is unsupported
            IOError: If file cannot be read
        """
        suffix = document_path.suffix.lower()

        if suffix == '.pdf':
            # Use PyMuPDF (fitz) for PDF extraction
            try:
                import fitz  # PyMuPDF
                doc = fitz.open(document_path)
                text_parts = []
                for page in doc:
                    text_parts.append(page.get_text())
                doc.close()
                text = '\n'.join(text_parts)

                # Normalize to NFC for deterministic hashing and span calculation
                text = unicodedata.normalize('NFC', text)
                logger.info(f"Extracted {len(text)} characters from PDF ({len(text_parts)}
pages)")
                return text
            except ImportError:
                logger.error("PyMuPDF (fitz) not available for PDF extraction")
                raise OSError(
                    "PDF extraction requires PyMuPDF (install with: pip install PyMuPDF).
"
                    "Alternatively, convert PDF to text manually."
                )
            except Exception as e:
                logger.error(f"Failed to extract PDF: {e}")
                raise OSError(f"PDF extraction failed: {e}")

        elif suffix in ['.txt', '.md']:
            # Plain text or markdown
            try:
                with open(document_path, encoding='utf-8') as f:
                    text = f.read()

                # Normalize to NFC for deterministic hashing and span calculation
                text = unicodedata.normalize('NFC', text)

                logger.info(f"Loaded {len(text)} characters from {suffix} file")
                return text
            except OSError as e:
                logger.error(f"Failed to read text file: {e}")
                raise

        else:
            raise ValueError(
                f"Unsupported file type: {suffix}. "
                f"Supported types: .pdf, .txt, .md"
            )

    async def process(
        self,
        document_path: Path,
        document_id: str = None,
        title: str = None
    ) -> CanonPolicyPackage:
        """
        Process a document through the complete SPC pipeline.

        Args:
            document_path: Path to input document
            document_id: Optional document identifier
            title: Optional document title
            max_chunks: Maximum number of chunks to generate

        Returns:
            CanonPolicyPackage: Orchestrator-ready policy package with:
                - ChunkGraph with all chunks and relationships
                - PolicyManifest with axes/programs/projects
                - QualityMetrics from SPC analysis
                - IntegrityIndex for verification
                - Rich SPC data preserved in metadata

        Raises:
            ValueError: If document is empty or invalid
            IOError: If document cannot be read
        """
        logger.info(f"Processing document: {document_path}")

        # Quality gate: Validate input file
        validation_input = self.quality_gates.validate_input(document_path)
        if not validation_input["passed"]:
            raise ValueError(
                f"SPC input validation failed: {validation_input['failures']}"
            )
        logger.info(f"Input validation passed (file size:
{validation_input['file_size_bytes']} bytes)")

        # Load document text (supports PDF, TXT, MD)
        try:
            document_text = self._load_document_text(document_path)
        except (OSError, ValueError) as e:
            logger.error(f"Failed to load document: {e}")
            raise

        if not document_text or not document_text.strip():
            raise ValueError(f"Document text is empty after extraction: {document_path}")

        logger.info(f"Document loaded: {len(document_text)} characters")

        # Prepare metadata
        metadata = {
            'document_id': document_id or str(document_path.stem),
            'title': title or document_path.name,
            'version': 'v3.0',
            'source_path': str(document_path)
        }

        # Process through chunking system (15-phase analysis)
        logger.info("Starting StrategicChunkingSystem.generate_smart_chunks()")
        smart_chunks = self.chunking_system.generate_smart_chunks(document_text, metadata)
        logger.info(f"Generated {len(smart_chunks)} SmartPolicyChunks")

        # Quality gate: Validate chunks
        chunk_dicts = [
            {
                "text": c.text,
                "chunk_id": c.chunk_id,
                "strategic_importance": c.strategic_importance,
                "quality_score": c.confidence_metrics.get("overall_confidence", get_parame
ter_loader().get("saaaaaa.processing.spc_ingestion.__init__.CPPIngestionPipeline._load_doc
ument_text").get("auto_param_L243_80", 0.0)),
            }
            for c in smart_chunks
        ]
        validation_chunks = self.quality_gates.validate_chunks(chunk_dicts)
        if not validation_chunks["passed"]:
            raise ValueError(
                f"SPC chunk validation failed: {validation_chunks['failures']}"
            )
        if validation_chunks.get("warnings"):
            logger.warning(f"Chunk validation warnings:
{validation_chunks['warnings'][:3]}")
        logger.info(f"Chunk validation passed ({validation_chunks['chunk_count']}
chunks)")


        # Convert to CanonPolicyPackage
        logger.info("Converting SmartPolicyChunks to CanonPolicyPackage")
        canon_package = self.converter.convert_to_canon_package(smart_chunks, metadata)

        # Log quality metrics
        if canon_package.quality_metrics:
            logger.info(
                f"Quality metrics - "
                f"provenance: {canon_package.quality_metrics.provenance_completeness:.2%},
 "
                f"coherence: {canon_package.quality_metrics.structural_consistency:.2%}, "
                f"coverage: {canon_package.quality_metrics.chunk_context_coverage:.2%}"
            )
        # RUNTIME VALIDATION: Validate CPP → Adapter contract
        if self.wiring_validator is not None:
            logger.info("Validating CPP → Adapter contract (runtime)")
            try:
                # Convert CanonPolicyPackage to dict for validation
                cpp_dict = self._canon_package_to_dict(canon_package)
                self.wiring_validator.validate_cpp_to_adapter(cpp_dict)
                logger.info("✓ CPP → Adapter contract validation passed")
            except Exception as e:
                logger.error(f"CPP → Adapter contract validation failed: {e}")
                raise ValueError(
                    f"Runtime contract violation at CPP → Adapter boundary: {e}"
                ) from e

        logger.info(f"Pipeline complete: {len(canon_package.chunk_graph.chunks)} chunks in
package")
        return canon_package

    def _canon_package_to_dict(self, canon_package: CanonPolicyPackage) -> dict[str, Any]:
        """Convert CanonPolicyPackage to dict for WiringValidator.

        Args:
            canon_package: CanonPolicyPackage to convert

        Returns:
            Dict representation for validation
        """
        # Extract chunks as list of dicts
        chunks = []
        if hasattr(canon_package, 'chunk_graph') and canon_package.chunk_graph:
            for chunk_id, chunk in canon_package.chunk_graph.chunks.items():
                chunk_dict = {
                    "chunk_id": chunk_id,
                    "text": chunk.text if hasattr(chunk, 'text') else "",
                    "text_span": {
                        "start": chunk.text_span.start if hasattr(chunk, 'text_span') else
0,
                        "end": chunk.text_span.end if hasattr(chunk, 'text_span') else 0,
                    } if hasattr(chunk, 'text_span') else {"start": 0, "end": 0},
                }
                chunks.append(chunk_dict)

        # Build validation dict
        return {
            "schema_version": canon_package.schema_version if hasattr(canon_package,
'schema_version') else "SPC-2025.1",
            "chunks": chunks,
            "chunk_count": len(chunks),
            "quality_metrics": {
                "provenance_completeness": (
                    canon_package.quality_metrics.provenance_completeness
                    if hasattr(canon_package, 'quality_metrics') and
canon_package.quality_metrics
                    else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.__in
it__.CPPIngestionPipeline._canon_package_to_dict").get("auto_param_L319_25", 0.0)
                ),
                "structural_consistency": (
                    canon_package.quality_metrics.structural_consistency
                    if hasattr(canon_package, 'quality_metrics') and
canon_package.quality_metrics
                    else get_parameter_loader().get("saaaaaa.processing.spc_ingestion.__in
it__.CPPIngestionPipeline._canon_package_to_dict").get("auto_param_L324_25", 0.0)
                ),
            } if hasattr(canon_package, 'quality_metrics') else {},
        }


__all__ = [
    'CPPIngestionPipeline',
    'StrategicChunkingSystem',
    'SmartChunkConverter',
]

===== FILE: src/saaaaaa/processing/spc_ingestion/converter.py =====
"""
SmartChunk to CanonPolicyPackage Converter
============================================

This module provides the critical bridge layer between the SPC (Smart Policy Chunks)
phase-one output and the CanonPolicyPackage format expected by the orchestrator.

Architecture:
    SmartPolicyChunk (from StrategicChunkingSystem)
        ↓
    SmartChunkConverter (this module)
        ↓
    CanonPolicyPackage (for SPCAdapter and Orchestrator)

Key Responsibilities:
1. Convert SmartPolicyChunk dataclass to Chunk dataclass
2. Map chunk_type (8 types) to resolution (MICRO/MESO/MACRO)
3. Extract policy/time/geo facets from SPC rich data
4. Build ChunkGraph with edges from related_chunks
5. Preserve SPC rich data in metadata for executor access
6. Generate quality metrics and integrity index
"""

from __future__ import annotations

import hashlib
import json
import logging
from typing import TYPE_CHECKING, Any

from saaaaaa.processing.cpp_ingestion.models import (
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method
    KPI,
    Budget,
    CanonPolicyPackage,
    Chunk,
    ChunkGraph,
    ChunkResolution,
    Confidence,
    Entity,
    GeoFacet,
    IntegrityIndex,
    PolicyFacet,
    PolicyManifest,
    ProvenanceMap,
    QualityMetrics,
    TextSpan,
    TimeFacet,
)

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.

        Args:
            smart_chunks: List of SmartPolicyChunk instances from StrategicChunkingSystem
            document_metadata: Document-level metadata (id, title, version, etc.)

        Returns:
            CanonPolicyPackage ready for orchestrator consumption

        Raises:
            ValueError: If smart_chunks is empty or invalid
        """
        # Defensive validation: ensure smart_chunks is non-empty
        if not smart_chunks or len(smart_chunks) == 0:
            raise ValueError(
                "Cannot convert empty smart_chunks list to CanonPolicyPackage. "
                "Minimum 1 chunk required from StrategicChunkingSystem."
            )

        # Defensive validation: check critical attributes on first chunk
        first_chunk = smart_chunks[0]
        required_attrs = ['chunk_id', 'document_id', 'text', 'document_position',
'chunk_type']
```

```
        missing_attrs = [attr for attr in required_attrs if not hasattr(first_chunk,
attr)]

        if missing_attrs:
            raise ValueError(
                f"SmartPolicyChunk missing critical attributes: {missing_attrs}. "
                f"Ensure StrategicChunkingSystem produced valid SmartPolicyChunk
instances. "
                f"Chunk type: {type(first_chunk)}"
            )

        self.logger.info(f"Converting {len(smart_chunks)} SmartPolicyChunks to
CanonPolicyPackage")

        # Build ChunkGraph
        chunk_graph = ChunkGraph()

        # Convert each SmartPolicyChunk to Chunk
        chunk_hashes = {}
        all_axes = set()
        all_programs = set()
        all_projects = set()
        all_years = set()
        all_territories = set()

        for smart_chunk in smart_chunks:
            # Convert to Chunk
            chunk = self._convert_smart_chunk_to_chunk(smart_chunk)

            # Add to ChunkGraph
            chunk_graph.chunks[chunk.id] = chunk
            chunk_hashes[chunk.id] = chunk.bytes_hash

            # Collect manifest data
            all_axes.update(chunk.policy_facets.axes)
            all_programs.update(chunk.policy_facets.programs)
            all_projects.update(chunk.policy_facets.projects)
            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("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._co
nvert_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)

        # Extract policy facets
        policy_facets = self._extract_policy_facets(smart_chunk)

        # Extract time facets
        time_facets = self._extract_time_facets(smart_chunk)

        # Extract geo facets
        geo_facets = self._extract_geo_facets(smart_chunk)

        # Build confidence from SPC metrics
        confidence = Confidence(
            layout=get_parameter_loader().get("saaaaaa.processing.spc_ingestion.converter.
SmartChunkConverter._convert_smart_chunk_to_chunk").get("auto_param_L232_19", 1.0),  # SPC
 doesn't distinguish these
            ocr=smart_chunk.confidence_metrics.get('extraction_confidence', get_parameter_
loader().get("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._convert_smar
t_chunk_to_chunk").get("auto_param_L233_76", 0.95)),
            typing=smart_chunk.coherence_score
        )

        # Create provenance
        provenance = self._build_provenance(smart_chunk)

        # Extract entities
        entities = self._extract_entities(smart_chunk)

        # Extract budget if available
        budget = self._extract_budget(smart_chunk)

        # Extract KPI if available
        kpi = self._extract_kpi(smart_chunk)

        # Build Chunk
        return Chunk(
            id=smart_chunk.chunk_id,
            text=smart_chunk.text,
            text_span=TextSpan(
                start=smart_chunk.document_position[0],
                end=smart_chunk.document_position[1]
            ),
            resolution=resolution,
            bytes_hash=smart_chunk.content_hash,
            policy_area_id=getattr(smart_chunk, 'policy_area_id', None),  # PA01-PA10
            dimension_id=getattr(smart_chunk, 'dimension_id', None),      # DIM01-DIM06
            policy_facets=policy_facets,
            time_facets=time_facets,
            geo_facets=geo_facets,
            confidence=confidence,
            provenance=provenance,
            budget=budget,
            kpi=kpi,
            entities=entities
        )

    @calibrated_method("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._ex
tract_policy_facets")
    def _extract_policy_facets(self, smart_chunk: Any) -> PolicyFacet:
        """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("saaaaaa.processing.spc_ingestion.converter.SmartChunkConverter._ex
tract_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)
```