```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 6
================================================================================
Generated: 2025-12-07T06:17:16.619183
Files in this batch: 17
================================================================================


================================================================================
FILE: src/farfan_pipeline/analysis/contradiction_deteccion.py
================================================================================

"""
Advanced Policy Contradiction Detection System for Colombian Municipal Development Plans

Este sistema implementa el estado del arte en detecciÃ³n de contradicciones para anÃ¡lisis
de polÃ-ticas pÃºblicas, especÃ-ficamente calibrado para Planes de Desarrollo Municipal (PDM)
colombianos segÃºn la Ley 152 de 1994 y metodologÃ-a DNP.

Innovations:
- Transformer-based semantic similarity using sentence-transformers
- Graph-based contradiction reasoning with NetworkX
- Bayesian inference for confidence scoring
- Temporal logic verification for timeline consistency
- Multi-dimensional vector embeddings for policy alignment
- Statistical hypothesis testing for numerical claims
"""

from __future__ import annotations

import logging
import re
from dataclasses import dataclass, field
from enum import Enum, auto
from typing import Any

import networkx as nx
import numpy as np
import torch
from scipy import stats
from scipy.spatial.distance import cosine
from scipy.stats import beta
from sentence_transformers import SentenceTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from transformers import AutoModelForSequenceClassification, DebertaV2Tokenizer, pipeline

# Check dependency lockdown
from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown

# Import runtime error fixes for defensive programming
from farfan_pipeline.utils.runtime_error_fixes import ensure_list_return, safe_text_extract
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

_lockdown = get_dependency_lockdown()
```

```

# Configure logging with structured format
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

class ContradictionType(Enum):
    """TaxonomÃa de contradicciones segÃºn estÃ¡ndares de polÃtica pÃºblica"""
    NUMERICAL_INCONSISTENCY = auto()
    TEMPORAL_CONFLICT = auto()
    SEMANTIC_OPPOSITION = auto()
    LOGICAL_INCOMPATIBILITY = auto()
    RESOURCE_ALLOCATION_MISMATCH = auto()
    OBJECTIVE_MISALIGNMENT = auto()
    REGULATORY_CONFLICT = auto()
    STAKEHOLDER_DIVERGENCE = auto()

class PolicyDimension(Enum):
    """Dimensiones del Plan de Desarrollo segÃºn DNP Colombia"""
    DIAGNOSTICO = "diagnÃ³stico"
    ESTRATEGICO = "estratÃ©gico"
    PROGRAMATICO = "programÃ¡tico"
    FINANCIERO = "plan plurianual de inversiones"
    SEGUIMIENTO = "seguimiento y evaluaciÃ³n"
    TERRITORIAL = "ordenamiento territorial"

@dataclass(frozen=True)
class PolicyStatement:
    """RepresentaciÃ³n estructurada de una declaraciÃ³n de polÃtica"""
    text: str
    dimension: PolicyDimension
    position: tuple[int, int]  # (start, end) in document
    entities: list[str] = field(default_factory=list)
    temporal_markers: list[str] = field(default_factory=list)
    quantitative_claims: list[dict[str, Any]] = field(default_factory=list)
    embedding: np.ndarray | None = None
    context_window: str = ""
    semantic_role: str | None = None
    dependencies: set[str] = field(default_factory=set)

@dataclass
class ContradictionEvidence:
    """Evidencia estructurada de contradicciÃ³n con trazabilidad completa"""
    statement_a: PolicyStatement
    statement_b: PolicyStatement
    contradiction_type: ContradictionType
    confidence: float  # Bayesian posterior probability
    severity: float  # Impact on policy coherence
    semantic_similarity: float
    logical_conflict_score: float
    temporal_consistency: bool
    numerical_divergence: float | None
    affected_dimensions: list[PolicyDimension]
    resolution_suggestions: list[str]
```

```
    graph_path: list[str] | None = None
    statistical_significance: float | None = None

class BayesianConfidenceCalculator:
    """
    Bayesian confidence calculator with domain-informed priors.

    Uses Beta distribution priors calibrated from empirical analysis of
    Colombian municipal development plans (PDMs).
    """

    def __init__(self) -> None:
        # Priors based on empirical analysis of Colombian municipal development plans (PDMs)
        self.prior_alpha = 2.5  # Shape parameter for beta distribution
        self.prior_beta = 7.5  # Scale parameter (conservative bias favoring lower confidence)

    def calculate_posterior(
            self,
            evidence_strength: float,
            observations: int,
            domain_weight: float = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L121_35", 1.0)
    ) -> float:
        """
        Calculate posterior probability using Bayesian inference.

        Updates the Beta distribution prior with observed evidence to compute
        the posterior mean, which represents the confidence level in the finding.

        Args:
            evidence_strength: Strength of the evidence (ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L130_57", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L130_61", 1.0) scale, unitless ratio)
            observations: Number of observations supporting the evidence (count)
            domain_weight: Policy domain-specific weight (multiplier, default: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L132_79", 1.0))

        Returns:
            float: Posterior probability (ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L135_42", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L135_46", 1.0) scale) representing confidence level
        """
        # Update Beta distribution with evidence
        alpha_post = self.prior_alpha + evidence_strength * observations * domain_weight
        beta_post = self.prior_beta + (1 - evidence_strength) * observations * domain_weight

        # Calculate mean of posterior distribution
        posterior_mean = alpha_post / (alpha_post + beta_post)

        # Calculate 95% credible interval
        credible_interval = beta.interval(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L145_42", 0.95), alpha_post, beta_post)

        # Adjust for uncertainty (wider intervals reduce confidence)
        uncertainty_penalty = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L148_30", 1.0) - (credible_interval[1] - credible_interval[0])
```

```
        return min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L150_19", 1.0), posterior_mean * uncertainty_penalty)

class TemporalLogicVerifier:
    """
    Temporal consistency verification using Linear Temporal Logic (LTL).

    Analyzes policy statements for temporal contradictions, deadline violations,
    and ordering conflicts using temporal logic patterns.
    """

    def __init__(self) -> None:
        self.temporal_patterns = {
            'sequential': re.compile(r'(primero|luego|despuÃ©s|posteriormente|finalmente)', re.IGNORECASE),
            'parallel': re.compile(r'(simultÃ¡neamente|al mismo tiempo|paralelamente)', re.IGNORECASE),
            'deadline': re.compile(r'(antes de|hasta|mÃ¡ximo|plazo)', re.IGNORECASE),
            'milestone': re.compile(r'(hito|meta intermedia|checkpoint)', re.IGNORECASE)
        }

    def verify_temporal_consistency(
            self,
            statements: list[PolicyStatement]
    ) -> tuple[bool, list[dict[str, Any]]]:
        """
        Verify temporal consistency between policy statements.

        Analyzes temporal ordering and deadline constraints to identify
        contradictions or violations in the policy timeline.

        Args:
            statements: List of policy statements to analyze

        Returns:
            tuple[bool, list[dict]]: A tuple containing:
                - is_consistent: True if no conflicts found
                - conflicts_found: List of detected temporal conflicts
        """
        timeline = self._build_timeline(statements)
        conflicts = []

        # Verify temporal ordering
        for i, event_a in enumerate(timeline):
            for event_b in timeline[i + 1:]:
                if self._has_temporal_conflict(event_a, event_b):
                    conflicts.append({
                        'event_a': event_a,
                        'event_b': event_b,
                        'conflict_type': 'temporal_ordering'
                    })

        # Verify deadline constraints
        deadline_violations = self._check_deadline_constraints(timeline)
        conflicts.extend(deadline_violations)

        return len(conflicts) == 0, conflicts
```

```

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._build_timeline")
    def _build_timeline(self, statements: list[PolicyStatement]) -> list[dict]:
        """
        Build timeline from policy statements.

        Extracts temporal markers and organizes them chronologically.

        Args:
            statements: List of policy statements

        Returns:
            list[dict]: Sorted timeline events with timestamps
        """
        timeline = []
        for stmt in statements:
            for marker in stmt.temporal_markers:
                # Extract structured temporal information
                timeline.append({
                    ’statement’: stmt,
                    ’marker’: marker,
                    ’timestamp’: self._parse_temporal_marker(marker),
                    ’type’: self._classify_temporal_type(marker)
                })
        return sorted(timeline, key=lambda x: x.get(’timestamp’, 0))

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._parse_temporal_marker")
    def _parse_temporal_marker(self, marker: str) -> int | None:
        """
        Parse temporal marker to numeric timestamp.

        Implements Colombian policy document temporal format parsing.

        Args:
            marker: Temporal marker string (e.g., "2024", "Q2", "segundo trimestre")

        Returns:
            int | None: Numeric timestamp, or None if parsing fails
        """
        # Implementation specific to Colombian policy document format
        year_match = re.search(r’20\d{2}’, marker)
        if year_match:
            return int(year_match.group())

        quarter_patterns = {
            ’primer’: 1, ’segundo’: 2, ’tercer’: 3, ’cuarto’: 4,
            ’Q1’: 1, ’Q2’: 2, ’Q3’: 3, ’Q4’: 4
        }
        for pattern, quarter in quarter_patterns.items():
            if pattern in marker.lower():
                return quarter

        return None

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._has_temporal_conflict")
    def _has_temporal_conflict(self, event_a: dict, event_b: dict) -> bool:
```

```
        """Detecta conflictos temporales entre eventos"""
        if event_a[’timestamp’] and event_b[’timestamp’]:
            # Verificar si eventos mutuamente excluyentes ocurren simultÃ¡neamente
            if event_a[’timestamp’] == event_b[’timestamp’]:
                return self._are_mutually_exclusive(
                    event_a[’statement’],
                    event_b[’statement’]
                )
        return False

    def _are_mutually_exclusive(
            self,
            stmt_a: PolicyStatement,
            stmt_b: PolicyStatement
    ) -> bool:
        """Determina si dos declaraciones son mutuamente excluyentes"""
        # Verificar si compiten por los mismos recursos
        resources_a = set(self._extract_resources(stmt_a.text))
        resources_b = set(self._extract_resources(stmt_b.text))

        return len(resources_a & resources_b) > 0

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._extract_resources")
    def _extract_resources(self, text: str) -> list[str]:
        """Extrae recursos mencionados en el texto"""
        resource_patterns = [
            r’presupuesto’,
            r’recursos?\s+\w+’,
            r’fondos?\s+\w+’,
            r’personal’,
            r’infraestructura’
        ]
        resources = []
        for pattern in resource_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            resources.extend(matches)
        return resources

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._check_deadline_constraints")
    def _check_deadline_constraints(self, timeline: list[dict]) -> list[dict]:
        """Verifica violaciones de restricciones de plazo"""
        violations = []
        for event in timeline:
            if event[’type’] == ’deadline’:
                # Verificar si hay eventos posteriores que deberÃ-an ocurrir antes
                for other in timeline:
                    if other[’timestamp’] and event[’timestamp’]:
                        if other[’timestamp’] > event[’timestamp’]:
                            if self._should_precede(other[’statement’], event[’statement’]):
                                violations.append({
                                    ’event_a’: other,
                                    ’event_b’: event,
                                    ’conflict_type’: ’deadline_violation’
                                })
        return violations

```

```
    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._should_precede")
    def _should_precede(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) -> bool:
        """Determina si stmt_a debe preceder a stmt_b"""
        # AnÃ¡lisis de dependencias causales
        return bool(stmt_a.dependencies & {stmt_b.text[:50]})

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._classify_temporal_type")
    def _classify_temporal_type(self, marker: str) -> str:
        """Clasifica el tipo de marcador temporal"""
        for pattern_type, pattern in self.temporal_patterns.items():
            if pattern.search(marker):
                return pattern_type
        return 'unspecified'

class PolicyContradictionDetector:
    """
    Sistema avanzado de detecciÃ³n de contradicciones para PDMs colombianos.
    Implementa el estado del arte en NLP y razonamiento lÃ³gico.
    """

    def __init__(
            self,
            model_name: str = "hiiamsid/sentence_similarity_spanish_es",
            spacy_model: str = "es_core_news_lg",
            device: str = "cuda" if torch.cuda.is_available() else "cpu"
    ) -> None:
        # Modelos de transformers para anÃ¡lisis semÃ¡ntico
        self.semantic_model = SentenceTransformer(model_name, device=device)

        # Modelo de clasificaciÃ³n de contradicciones
        model_name = "microsoft/deberta-v3-base"
        tokenizer = DebertaV2Tokenizer.from_pretrained(model_name)
        model = AutoModelForSequenceClassification.from_pretrained(model_name)

        self.contradiction_classifier = pipeline(
            "text-classification",
            model=model,
            tokenizer=tokenizer,
            device=0 if device == "cuda" else -1,
        )

        # Procesamiento de lenguaje natural
        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import load_spacy_model
        self.nlp = load_spacy_model(spacy_model)

        # Componentes especializados
        self.bayesian_calculator = BayesianConfidenceCalculator()
        self.temporal_verifier = TemporalLogicVerifier()

        # Grafo de conocimiento para razonamiento
        self.knowledge_graph = nx.DiGraph()

        # Vectorizador TF-IDF para anÃ¡lisis complementario
        self.tfidf = TfidfVectorizer(
            ngram_range=(1, 3),
```

```
            max_features=5000,
            sublinear_tf=True
        )

        # Patrones especÃficos de PDM colombiano
        self._initialize_pdm_patterns()

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._initialize_pdm_patterns")
    def _initialize_pdm_patterns(self) -> None:
        """Inicializa patrones especÃficos de PDMs colombianos"""
        self.pdm_patterns = {
            'ejes_estrategicos': re.compile(
                r'(eje\s+estratÃ©gico|lÃnea\s+estratÃ©gica|pilar|dimensiÃ³n)',
                re.IGNORECASE
            ),
            'programas': re.compile(
                r'(programa|subprograma|proyecto|iniciativa)',
                re.IGNORECASE
            ),
            'metas': re.compile(
                r'(meta\s+de\s+resultado|meta\s+de\s+producto|indicador)',
                re.IGNORECASE
            ),
            'recursos': re.compile(
                r'(SGP|regalÃas|recursos\s+propios|cofinanciaciÃ³n|crÃ©dito)',
                re.IGNORECASE
            ),
            'normativa': re.compile(
                r'(ley\s+\d+|decreto\s+\d+|acuerdo\s+\d+|resoluciÃ³n\s+\d+)',
                re.IGNORECASE
            )
        }

    def detect(
            self,
            text: str,
            plan_name: str = "PDM",
            dimension: PolicyDimension = PolicyDimension.ESTRATEGICO
    ) -> dict[str, Any]:
        """
        Detecta contradicciones con anÃ¡lisis multi-dimensional avanzado

        Args:
            text: Texto del plan de desarrollo
            plan_name: Nombre del PDM
            dimension: DimensiÃ³n del plan siendo analizada

        Returns:
            AnÃ¡lisis completo con contradicciones detectadas y mÃ©tricas
        """
        # Extraer declaraciones de polÃtica estructuradas
        statements = self._extract_policy_statements(text, dimension)

        # Generar embeddings semÃ¡nticos
        statements = self._generate_embeddings(statements)

```

```
        # Construir grafo de conocimiento
        self._build_knowledge_graph(statements)

        # Detectar contradicciones multi-tipo
        contradictions = []

        # 1. Contradicciones semÃ¡nticas usando transformers
        semantic_contradictions = self._detect_semantic_contradictions(statements)
        contradictions.extend(ensure_list_return(semantic_contradictions))

        # 2. Inconsistencias numÃ©ricas con pruebas estadÃsticas
        numerical_contradictions = self._detect_numerical_inconsistencies(statements)
        contradictions.extend(ensure_list_return(numerical_contradictions))

        # 3. Conflictos temporales con verificaciÃ³n lÃ³gica
        temporal_conflicts = self._detect_temporal_conflicts(statements)
        contradictions.extend(ensure_list_return(temporal_conflicts))

        # 4. Incompatibilidades lÃ³gicas usando razonamiento en grafo
        logical_contradictions = self._detect_logical_incompatibilities(statements)
        contradictions.extend(ensure_list_return(logical_contradictions))

        # 5. Conflictos de asignaciÃ³n de recursos
        resource_conflicts = self._detect_resource_conflicts(statements)
        contradictions.extend(ensure_list_return(resource_conflicts))

        # Calcular mÃ©tricas agregadas
        coherence_metrics = self._calculate_coherence_metrics(
            contradictions,
            statements,
            text
        )

        # Generar recomendaciones de resoluciÃ³n
        recommendations = self._generate_resolution_recommendations(contradictions)

        return {
            "plan_name": plan_name,
            "dimension": dimension.value,
            "contradictions": [self._serialize_contradiction(c) for c in contradictions],
            "total_contradictions": len(contradictions),
            "high_severity_count": sum(1 for c in contradictions if c.severity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._initialize_pdm_patterns", "auto_param_L469_81", 0.7)),
            "coherence_metrics": coherence_metrics,
            "recommendations": recommendations,
            "knowledge_graph_stats": self._get_graph_statistics()
        }

    def _extract_policy_statements(
            self,
            text: str,
            dimension: PolicyDimension
    ) -> list[PolicyStatement]:
        """Extrae declaraciones de polÃtica estructuradas del texto"""
        doc = self.nlp(text)
        statements = []
```

```

        for sent in doc.sents:
            # Analizar entidades nombradas
            entities = [ent.text for ent in sent.ents]

            # Extraer marcadores temporales
            temporal_markers = self._extract_temporal_markers(sent.text)

            # Extraer afirmaciones cuantitativas
            quantitative_claims = self._extract_quantitative_claims(sent.text)

            # Determinar rol semÃ¡ntico
            semantic_role = self._determine_semantic_role(sent)

            # Identificar dependencias
            dependencies = self._identify_dependencies(sent, doc)

            statement = PolicyStatement(
                text=sent.text,
                dimension=dimension,
                position=(sent.start_char, sent.end_char),
                entities=entities,
                temporal_markers=temporal_markers,
                quantitative_claims=quantitative_claims,
                context_window=self._get_context_window(text, sent.start_char, sent.end_char),
                semantic_role=semantic_role,
                dependencies=dependencies
            )

            statements.append(statement)

        return statements

    def _generate_embeddings(
            self,
            statements: list[PolicyStatement]
    ) -> list[PolicyStatement]:
        """Genera embeddings semÃ¡nticos para las declaraciones"""
        texts = [stmt.text for stmt in statements]
        embeddings = self.semantic_model.encode(texts, convert_to_numpy=True)

        # Crear nuevas instancias con embeddings
        enhanced_statements = []
        for stmt, embedding in zip(statements, embeddings, strict=False):
            enhanced = PolicyStatement(
                text=stmt.text,
                dimension=stmt.dimension,
                position=stmt.position,
                entities=stmt.entities,
                temporal_markers=stmt.temporal_markers,
                quantitative_claims=stmt.quantitative_claims,
                embedding=embedding,
                context_window=stmt.context_window,
                semantic_role=stmt.semantic_role,
                dependencies=stmt.dependencies
            )
```

```
                enhanced_statements.append(enhanced)

        return enhanced_statements

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph")
    def _build_knowledge_graph(self, statements: list[PolicyStatement]) -> None:
        """Construye grafo de conocimiento para razonamiento"""
        self.knowledge_graph.clear()

        for i, stmt in enumerate(statements):
            node_id = f"stmt_{i}"
            self.knowledge_graph.add_node(
                node_id,
                text=stmt.text[:100],
                dimension=stmt.dimension.value,
                entities=stmt.entities,
                semantic_role=stmt.semantic_role
            )

            # Conectar con declaraciones relacionadas
            for j, other in enumerate(statements):
                if i != j:
                    similarity = self._calculate_similarity(stmt, other)
                    if similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L562_36", 0.7):  # Umbral de relaciÃ³n
                        self.knowledge_graph.add_edge(
                            f"stmt_{i}",
                            f"stmt_{j}",
                            weight=similarity,
                            relation_type=self._determine_relation_type(stmt, other)
                        )

    def _detect_semantic_contradictions(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta contradicciones semÃ¡nticas usando transformers"""
        contradictions = []

        for i, stmt_a in enumerate(statements):
            for stmt_b in statements[i + 1:]:
                if stmt_a.embedding is not None and stmt_b.embedding is not None:
                    # Calcular similaridad coseno
                    similarity = 1 - cosine(stmt_a.embedding, stmt_b.embedding)

                    # Verificar contradicciÃ³n usando clasificador
                    combined_text = f"{stmt_a.text} [SEP] {stmt_b.text}"
                    contradiction_score = self._classify_contradiction(combined_text)

                    if contradiction_score > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L587_45", 0.7) and similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L587_66", 0.5):
                        # Calcular confianza Bayesiana
                        confidence = self.bayesian_calculator.calculate_posterior(
                            evidence_strength=contradiction_score,
                            observations=len(stmt_a.entities) + len(stmt_b.entities),
```

```
                            domain_weight=self._get_domain_weight(stmt_a.dimension)
                        )

                        evidence = ContradictionEvidence(
                            statement_a=stmt_a,
                            statement_b=stmt_b,
                            contradiction_type=ContradictionType.SEMANTIC_OPPOSITION,
                            confidence=confidence,
                            severity=self._calculate_severity(stmt_a, stmt_b),
                            semantic_similarity=similarity,
                            logical_conflict_score=contradiction_score,
                            temporal_consistency=True,
                            numerical_divergence=None,
                            affected_dimensions=[stmt_a.dimension, stmt_b.dimension],
                            resolution_suggestions=self._suggest_resolutions(
                                ContradictionType.SEMANTIC_OPPOSITION
                            )
                        )
                        contradictions.append(evidence)

        return contradictions

    def _detect_numerical_inconsistencies(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta inconsistencias numÃ©ricas con anÃ¡lisis estadÃstico"""
        contradictions = []

        for i, stmt_a in enumerate(statements):
            for stmt_b in statements[i + 1:]:
                if stmt_a.quantitative_claims and stmt_b.quantitative_claims:
                    for claim_a in stmt_a.quantitative_claims:
                        for claim_b in stmt_b.quantitative_claims:
                            if self._are_comparable_claims(claim_a, claim_b):
                                divergence = self._calculate_numerical_divergence(
                                    claim_a,
                                    claim_b
                                )

                                if divergence is not None and divergence > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L632_75", 0.2):
                                    # Test estadÃstico de significancia
                                    p_value = self._statistical_significance_test(
                                        claim_a,
                                        claim_b
                                    )

                                    if p_value < ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L639_49", 0.05):  # Significancia estadÃstica
                                        confidence = self.bayesian_calculator.calculate_posterior(
                                            evidence_strength=1 - p_value,
                                            observations=2,
                                            domain_weight=1.5  # Mayor peso para evidencia numÃ©rica
                                        )

```

```
                                        evidence = ContradictionEvidence(
                                            statement_a=stmt_a,
                                            statement_b=stmt_b,
                                            contradiction_type=ContradictionType.NUMERICAL_INCONSISTENCY,
                                            confidence=confidence,
                                            severity=min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L651_57", 1.0), divergence),
                                            semantic_similarity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L652_64", 0.0),
                                            logical_conflict_score=divergence,
                                            temporal_consistency=True,
                                            numerical_divergence=divergence,
                                            affected_dimensions=[stmt_a.dimension],
                                            resolution_suggestions=self._suggest_resolutions(
                                                ContradictionType.NUMERICAL_INCONSISTENCY
                                            ),
                                            statistical_significance=p_value
                                        )
                                        contradictions.append(evidence)

        return contradictions

    def _detect_temporal_conflicts(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta conflictos temporales usando verificaciÃ³n lÃ³gica"""
        contradictions = []

        # Filtrar declaraciones con marcadores temporales
        temporal_statements = [s for s in statements if s.temporal_markers]

        if len(temporal_statements) >= 2:
            is_consistent, conflicts = self.temporal_verifier.verify_temporal_consistency(
                temporal_statements
            )

            for conflict in conflicts:
                stmt_a = conflict['event_a']['statement']
                stmt_b = conflict['event_b']['statement']

                confidence = self.bayesian_calculator.calculate_posterior(
                    evidence_strength=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L686_38", 0.9),  # Alta confianza en lÃ³gica temporal
                    observations=len(conflicts),
                    domain_weight=1.2
                )

                evidence = ContradictionEvidence(
                    statement_a=stmt_a,
                    statement_b=stmt_b,
                    contradiction_type=ContradictionType.TEMPORAL_CONFLICT,
                    confidence=confidence,
                    severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L696_29", 0.8),  # Los conflictos temporales son severos
                    semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
```

```
                    logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L698_43", 1.0),
                    temporal_consistency=False,
                    numerical_divergence=None,
                    affected_dimensions=[PolicyDimension.PROGRAMATICO],
                    resolution_suggestions=self._suggest_resolutions(
                        ContradictionType.TEMPORAL_CONFLICT
                    )
                )
                contradictions.append(evidence)

        return contradictions

    def _detect_logical_incompatibilities(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta incompatibilidades lÃ³gicas usando razonamiento en grafo"""
        contradictions = []

        # Buscar ciclos negativos en el grafo (indicativos de contradicciÃ³n)
        try:
            negative_cycles = nx.negative_edge_cycle(
                self.knowledge_graph,
                weight='weight'
            )

            for cycle in negative_cycles:
                # Extraer declaraciones del ciclo
                stmt_indices = [int(node.split('_')[1]) for node in cycle]
                cycle_statements = [statements[i] for i in stmt_indices]

                # Analizar incompatibilidad lÃ³gica
                for i in range(len(cycle_statements)):
                    stmt_a = cycle_statements[i]
                    stmt_b = cycle_statements[(i + 1) % len(cycle_statements)]

                    if self._has_logical_conflict(stmt_a, stmt_b):
                        confidence = self.bayesian_calculator.calculate_posterior(
                            evidence_strength=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L736_46", 0.85),
                            observations=len(cycle),
                            domain_weight = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "domain_weight", 1.0) # Refactored
                        )

                        evidence = ContradictionEvidence(
                            statement_a=stmt_a,
                            statement_b=stmt_b,
                            contradiction_type=ContradictionType.LOGICAL_INCOMPATIBILITY,
                            confidence=confidence,
                            severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L746_37", 0.7),
                            semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
                            logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L748_51", 0.9),
```

```
                            temporal_consistency=True,
                            numerical_divergence=None,
                            affected_dimensions=[stmt_a.dimension, stmt_b.dimension],
                            resolution_suggestions=self._suggest_resolutions(
                                ContradictionType.LOGICAL_INCOMPATIBILITY
                            ),
                            graph_path=cycle
                        )
                        contradictions.append(evidence)
        except nx.NetworkXError:
            pass  # No negative cycles found

        return contradictions

    def _detect_resource_conflicts(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta conflictos en asignaciÃ³n de recursos"""
        contradictions = []
        resource_allocations = {}

        for stmt in statements:
            # Extraer menciones de recursos
            resources = self._extract_resource_mentions(stmt.text)
            for resource_type, amount in resources:
                if resource_type not in resource_allocations:
                    resource_allocations[resource_type] = []
                resource_allocations[resource_type].append((stmt, amount))

        # Verificar conflictos de asignaciÃ³n
        for resource_type, allocations in resource_allocations.items():
            if len(allocations) > 1:
                total_claimed = sum(amount for _, amount in allocations if amount)

                # Verificar si las asignaciones son mutuamente excluyentes
                for i, (stmt_a, amount_a) in enumerate(allocations):
                    for stmt_b, amount_b in allocations[i + 1:]:
                        if amount_a and amount_b:
                            if self._are_conflicting_allocations(
                                    amount_a,
                                    amount_b,
                                    total_claimed
                            ):
                                confidence = self.bayesian_calculator.calculate_posterior(
                                    evidence_strength=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L794_54", 0.8),
                                    observations=len(allocations),
                                    domain_weight=1.3
                                )

                                evidence = ContradictionEvidence(
                                    statement_a=stmt_a,
                                    statement_b=stmt_b,
                                    contradiction_type=ContradictionType.RESOURCE_ALLOCATION_MISMATCH,
                                    confidence=confidence,
```

```
                                    severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L804_45", 0.9),  # Conflictos de recursos son crÃ-ticos
                                    semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
                                    logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L806_59", 0.8),
                                    temporal_consistency=True,
                                    numerical_divergence=abs(amount_a - amount_b) / max(amount_a, amount_b),
                                    affected_dimensions=[PolicyDimension.FINANCIERO],
                                    resolution_suggestions=self._suggest_resolutions(
                                        ContradictionType.RESOURCE_ALLOCATION_MISMATCH
                                    )
                                )
                                contradictions.append(evidence)

        return contradictions

    def _calculate_coherence_metrics(
            self,
            contradictions: list[ContradictionEvidence],
            statements: list[PolicyStatement],
            text: str
    ) -> dict[str, float]:
        """Calcula mÃ©tricas avanzadas de coherencia del documento"""

        # Densidad de contradicciones normalizada
        contradiction_density = len(contradictions) / max(1, len(statements))

        # Ã\215ndice de coherencia semÃ¡ntica global
        semantic_coherence = self._calculate_global_semantic_coherence(statements)

        # Consistencia temporal
        temporal_consistency = sum(
            1 for c in contradictions
            if c.contradiction_type != ContradictionType.TEMPORAL_CONFLICT
        ) / max(1, len(contradictions))

        # AlineaciÃ³n de objetivos
        objective_alignment = self._calculate_objective_alignment(statements)

        # Ã\215ndice de fragmentaciÃ³n del grafo
        graph_fragmentation = self._calculate_graph_fragmentation()

        # Score de coherencia compuesto (weighted harmonic mean)
        weights = np.array([ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_28", 0.3), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_33", 0.25), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_39", 0.2), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_44", 0.15), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_50", 0.1)])
        scores = np.array([
            1 - contradiction_density,
            semantic_coherence,
            temporal_consistency,
            objective_alignment,
            1 - graph_fragmentation
        ])

```

```
        # Harmonic mean ponderada para penalizar valores bajos
        coherence_score = np.sum(weights) / np.sum(weights / np.maximum(scores, ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L855_80", 0.01)))

        # EntropÃ-a de contradicciones
        contradiction_entropy = self._calculate_contradiction_entropy(contradictions)

        # Complejidad sintÃ¡ctica del documento
        syntactic_complexity = self._calculate_syntactic_complexity(text)

        return {
            "coherence_score": float(coherence_score),
            "contradiction_density": float(contradiction_density),
            "semantic_coherence": float(semantic_coherence),
            "temporal_consistency": float(temporal_consistency),
            "objective_alignment": float(objective_alignment),
            "graph_fragmentation": float(graph_fragmentation),
            "contradiction_entropy": float(contradiction_entropy),
            "syntactic_complexity": float(syntactic_complexity),
            "confidence_interval": self._calculate_confidence_interval(coherence_score, len(statements))
        }

    def _calculate_global_semantic_coherence(
            self,
            statements: list[PolicyStatement]
    ) -> float:
        """Calcula coherencia semÃ¡ntica global usando embeddings"""
        if len(statements) < 2:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L881_19", 1.0)

        # Calcular matriz de similitud
        embeddings = [s.embedding for s in statements if s.embedding is not None]
        if len(embeddings) < 2:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L886_19", 0.5)

        similarity_matrix = cosine_similarity(embeddings)

        # Calcular coherencia como promedio de similitudes consecutivas
        consecutive_similarities = []
        for i in range(len(similarity_matrix) - 1):
            consecutive_similarities.append(similarity_matrix[i, i + 1])

        # Penalizar alta varianza en similitudes
        mean_similarity = np.mean(consecutive_similarities)
        std_similarity = np.std(consecutive_similarities)

        coherence = mean_similarity * (1 - min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L899_47", 0.5), std_similarity))

        return float(coherence)

    def _calculate_objective_alignment(
            self,
            statements: list[PolicyStatement]
```

```
    ) -> float:
        """Calcula alineaciÃ³n entre objetivos declarados"""
        objective_statements = [
            s for s in statements
            if s.semantic_role in ['objective', 'goal', 'target']
        ]

        if len(objective_statements) < 2:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L914_19", 1.0)

        # Analizar consistencia direccional de objetivos
        alignment_scores = []
        for i, obj_a in enumerate(objective_statements):
            for obj_b in objective_statements[i + 1:]:
                if obj_a.embedding is not None and obj_b.embedding is not None:
                    # Calcular alineaciÃ³n como similitud coseno
                    alignment = 1 - cosine(obj_a.embedding, obj_b.embedding)
                    alignment_scores.append(alignment)

        if alignment_scores:
            return float(np.mean(alignment_scores))
        return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L927_15", 0.5)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation")
    def _calculate_graph_fragmentation(self) -> float:
        """Calcula fragmentaciÃ³n del grafo de conocimiento"""
        if self.knowledge_graph.number_of_nodes() == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation", "auto_param_L933_19", 0.0)

        # Calcular nÃºmero de componentes conectados
        num_components = nx.number_weakly_connected_components(self.knowledge_graph)
        num_nodes = self.knowledge_graph.number_of_nodes()

        # FragmentaciÃ³n normalizada
        fragmentation = (num_components - 1) / max(1, num_nodes - 1)

        return float(fragmentation)

    def _calculate_contradiction_entropy(
            self,
            contradictions: list[ContradictionEvidence]
    ) -> float:
        """Calcula entropÃa de distribuciÃ³n de tipos de contradicciÃ³n"""
        if not contradictions:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation", "auto_param_L950_19", 0.0)

        # Contar frecuencia de cada tipo
        type_counts = {}
        for c in contradictions:
            type_counts[c.contradiction_type] = type_counts.get(c.contradiction_type, 0) + 1

        # Calcular probabilidades
```

```
        total = len(contradictions)
        probabilities = [count / total for count in type_counts.values()]

        # Calcular entropÃa de Shannon
        entropy = -sum(p * np.log2(p) if p > 0 else 0 for p in probabilities)

        # Normalizar por entropÃa mÃ¡xima
        max_entropy = np.log2(len(ContradictionType))
        normalized_entropy = entropy / max_entropy if max_entropy > 0 else 0

        return float(normalized_entropy)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity")
    def _calculate_syntactic_complexity(self, text: str) -> float:
        """Calcula complejidad sintÃ¡ctica del documento"""
        doc = self.nlp(text[:5000])  # Limitar para eficiencia

        # MÃ©tricas de complejidad
        avg_sentence_length = np.mean([len(sent.text.split()) for sent in doc.sents])

        # Profundidad promedio del Ã¡rbol de dependencias
        dependency_depths = []
        for sent in doc.sents:
            depths = [self._get_dependency_depth(token) for token in sent]
            if depths:
                dependency_depths.append(np.mean(depths))

        avg_dependency_depth = np.mean(dependency_depths) if dependency_depths else 0

        # Diversidad lÃ©xica (Type-Token Ratio)
        tokens = [token.text.lower() for token in doc if token.is_alpha]
        ttr = len(set(tokens)) / len(tokens) if tokens else 0

        # Combinar mÃ©tricas
        complexity = (
                min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L993_20", 1.0), avg_sentence_length / 50) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L993_53", 0.3) +
                min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L994_20", 1.0), avg_dependency_depth / 10) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L994_54", 0.3) +
                ttr * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L995_22", 0.4)
        )

        return float(complexity)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth")
    def _get_dependency_depth(self, token) -> int:
        """Calcula profundidad de un token en el Ã¡rbol de dependencias"""
        depth = 0
        current = token
        while current.head != current and depth < 20:  # Evitar loops infinitos
            current = current.head
            depth += 1
        return depth
```

```

    def _calculate_confidence_interval(
            self,
            score: float,
            n_observations: int
    ) -> tuple[float, float]:
        """Calcula intervalo de confianza del 95% para el score"""
        # Usar distribuciÃ³n t de Student para muestras pequeÃ±as
        if n_observations < 30:
            # Error estÃ¡ndar estimado
            se = np.sqrt(score * (1 - score) / n_observations)
            # Valor crÃtico t para 95% de confianza
            t_critical = stats.t.ppf(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth", "auto_param_L1021_37", 0.975), n_observations - 1)
            margin = t_critical * se
        else:
            # Usar distribuciÃ³n normal para muestras grandes
            se = np.sqrt(score * (1 - score) / n_observations)
            margin = 1.96 * se

        return (
            max(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth", "auto_param_L1029_16", 0.0), score - margin),
            min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth", "auto_param_L1030_16", 1.0), score + margin)
        )

    def _generate_resolution_recommendations(
            self,
            contradictions: list[ContradictionEvidence]
    ) -> list[dict[str, Any]]:
        """Genera recomendaciones especÃficas para resolver contradicciones"""
        recommendations = []

        # Agrupar contradicciones por tipo
        by_type = {}
        for c in contradictions:
            if c.contradiction_type not in by_type:
                by_type[c.contradiction_type] = []
            by_type[c.contradiction_type].append(c)

        # Generar recomendaciones por tipo
        for cont_type, conflicts in by_type.items():
            if cont_type == ContradictionType.NUMERICAL_INCONSISTENCY:
                recommendations.append({
                    "type": "numerical_reconciliation",
                    "priority": "high",
                    "description": "Revisar y reconciliar cifras inconsistentes",
                    "specific_actions": [
                        "Verificar fuentes de datos originales",
                        "Establecer lÃnea base Ãºnica",
                        "Documentar metodologÃa de cÃ¡lculo"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

```

```
            elif cont_type == ContradictionType.TEMPORAL_CONFLICT:
                recommendations.append({
                    "type": "timeline_adjustment",
                    "priority": "high",
                    "description": "Ajustar cronograma para resolver conflictos temporales",
                    "specific_actions": [
                        "Revisar secuencia de actividades",
                        "Validar plazos con Ã¡reas responsables",
                        "Establecer hitos intermedios claros"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

            elif cont_type == ContradictionType.RESOURCE_ALLOCATION_MISMATCH:
                recommendations.append({
                    "type": "budget_reallocation",
                    "priority": "critical",
                    "description": "Revisar asignaciÃ³n presupuestal",
                    "specific_actions": [
                        "Realizar anÃ¡lisis de suficiencia presupuestal",
                        "Priorizar programas segÃºn impacto",
                        "Identificar fuentes alternativas de financiaciÃ³n"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

            elif cont_type == ContradictionType.SEMANTIC_OPPOSITION:
                recommendations.append({
                    "type": "conceptual_clarification",
                    "priority": "medium",
                    "description": "Clarificar conceptos y objetivos opuestos",
                    "specific_actions": [
                        "Realizar sesiones de alineaciÃ³n estratÃ©gica",
                        "Definir glosario de tÃ©rminos unificado",
                        "Establecer jerarquÃa clara de objetivos"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

        # Ordenar por prioridad
        priority_order = {"critical": 0, "high": 1, "medium": 2, "low": 3}
        recommendations.sort(key=lambda x: priority_order.get(x["priority"], 4))

        return recommendations

    def _identify_affected_sections(
            self,
            conflicts: list[ContradictionEvidence]
    ) -> list[str]:
        """Identifica secciones del plan afectadas por contradicciones"""
        affected = set()
        for c in conflicts:
            # Extraer informaciÃ³n de secciÃ³n desde el contexto
            for pattern_name, pattern in self.pdm_patterns.items():
                if pattern.search(c.statement_a.context_window):
                    affected.add(pattern_name)
```

```
                if pattern.search(c.statement_b.context_window):
                    affected.add(pattern_name)

        return list(affected)

    def _serialize_contradiction(
            self,
            contradiction: ContradictionEvidence
    ) -> dict[str, Any]:
        """Serializa evidencia de contradicciÃ³n para output"""
        return {
            "statement_1": contradiction.statement_a.text,
            "statement_2": contradiction.statement_b.text,
            "position_1": contradiction.statement_a.position,
            "position_2": contradiction.statement_b.position,
            "contradiction_type": contradiction.contradiction_type.name,
            "confidence": float(contradiction.confidence),
            "severity": float(contradiction.severity),
            "semantic_similarity": float(contradiction.semantic_similarity),
            "logical_conflict_score": float(contradiction.logical_conflict_score),
            "temporal_consistency": contradiction.temporal_consistency,
            "numerical_divergence": float(
                contradiction.numerical_divergence) if contradiction.numerical_divergence else None,
            "statistical_significance": float(
                contradiction.statistical_significance) if contradiction.statistical_significance else None,
            "affected_dimensions": [d.value for d in contradiction.affected_dimensions],
            "resolution_suggestions": contradiction.resolution_suggestions,
            "graph_path": contradiction.graph_path
        }

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_graph_statistics")
    def _get_graph_statistics(self) -> dict[str, Any]:
        """Obtiene estadÃsticas del grafo de conocimiento"""
        if self.knowledge_graph.number_of_nodes() == 0:
            return {"nodes": 0, "edges": 0, "components": 0}

        return {
            "nodes": self.knowledge_graph.number_of_nodes(),
            "edges": self.knowledge_graph.number_of_edges(),
            "components": nx.number_weakly_connected_components(self.knowledge_graph),
            "density": nx.density(self.knowledge_graph),
            "average_clustering": nx.average_clustering(self.knowledge_graph.to_undirected()),
            "diameter": nx.diameter(self.knowledge_graph.to_undirected()) if nx.is_connected(
                self.knowledge_graph.to_undirected()) else -1
        }

    # MÃ©todos auxiliares

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_temporal_markers")
    def _extract_temporal_markers(self, text: str) -> list[str]:
        """Extrae marcadores temporales del texto"""
        markers = []

        # Patrones de fechas
        date_patterns = [
            r'\d{1,2}\s+de\s+\w+\s+de\s+\d{4}',
```

```
            r'\d{4}-\d{2}-\d{2}',
            r'(enero|febrero|marzo|abril|mayo|junio|julio|agosto|septiembre|octubre|noviembre|diciembre)\s+\d{4}',
            r'(Q[1-4]|trimestre\s+[1-4])\s+\d{4}',
            r'20\d{2}',
            r'(corto|mediano|largo)\s+plazo',
            r'(primer|segundo|tercer|cuarto)\s+(aÃ±o|semestre|trimestre)'
        ]

        for pattern in date_patterns:
            matches = re.findall(pattern, text, re.IGNORECASE)
            markers.extend(matches)

        return markers

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_quantitative_claims")
    def _extract_quantitative_claims(self, text: str) -> list[dict[str, Any]]:
        """Extrae afirmaciones cuantitativas estructuradas"""
        claims = []

        # Patrones numÃ©ricos con contexto
        patterns = [
            (r'(\d+(?:[.,]\d+)?)\s*(%|por\s*ciento)', 'percentage'),
            (r'(\d+(?:[.,]\d+)?)\s*(millones?|mil\s+millones?)', 'amount'),
            (r'(\$|COP)\s*(\d+(?:[.,]\d+)?)', 'currency'),
            (r'(\d+(?:[.,]\d+)?)\s*(personas?|beneficiarios?|familias?)', 'beneficiaries'),
            (r'(\d+(?:[.,]\d+)?)\s*(hectÃ¡reas?|km2?|metros?)', 'area'),
            (r'meta\s+de\s+(\d+(?:[.,]\d+)?)', 'target')
        ]

        for pattern, claim_type in patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                value_str = match.group(1) if claim_type != 'currency' else match.group(2)
                value = self._parse_number(value_str)

                claims.append({
                    'type': claim_type,
                    'value': value,
                    'raw_text': match.group(0),
                    'position': match.span(),
                    'context': text[max(0, match.start() - 20):min(len(text), match.end() + 20)]
                })

        return claims

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._parse_number")
    def _parse_number(self, text: str) -> float:
        """Parsea nÃºmero desde texto"""
        try:
            # Reemplazar coma decimal
            normalized = text.replace(',', '.')
            return float(normalized)
        except ValueError:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._parse_number", "auto_param_L1227_19", 0.0)

```

```
    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_resource_mentions")
    def _extract_resource_mentions(self, text: str) -> list[tuple[str, float | None]]:
        """Extrae menciones de recursos con montos"""
        resources = []

        # Patrones de recursos especÃ-ficos de PDM colombiano
        resource_patterns = [
            (r'SGP\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'SGP'),
            (r'regalÃ-as\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'regalÃ-as'),
            (r'recursos\s+propios\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'recursos_propios'),
            (r'cofinanciaciÃ³n\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'cofinanciaciÃ³n'),
            (r'presupuesto\s+total\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'presupuesto_total')
        ]

        for pattern, resource_type in resource_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                amount = self._parse_number(match.group(1)) if match.group(1) else None
                if match.group(2) and 'millon' in match.group(2).lower():
                    amount = amount * 1000000 if amount else None
                resources.append((resource_type, amount))

        return resources

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._determine_semantic_role")
    def _determine_semantic_role(self, sent) -> str | None:
        """Determina el rol semÃ¡ntico de una oraciÃ³n"""
        # Safely extract text (handles both strings and spacy objects)
        text_lower = safe_text_extract(sent).lower()

        role_patterns = {
            'objective': ['objetivo', 'meta', 'propÃ³sito', 'finalidad'],
            'strategy': ['estrategia', 'lÃ-nea', 'eje', 'pilar'],
            'action': ['implementar', 'ejecutar', 'desarrollar', 'realizar'],
            'indicator': ['indicador', 'medir', 'evaluar', 'monitorear'],
            'resource': ['presupuesto', 'recurso', 'financiaciÃ³n', 'inversiÃ³n'],
            'constraint': ['limitaciÃ³n', 'restricciÃ³n', 'condiciÃ³n', 'requisito']
        }

        for role, keywords in role_patterns.items():
            if any(keyword in text_lower for keyword in keywords):
                return role

        return None

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._identify_dependencies")
    def _identify_dependencies(self, sent, doc) -> set[str]:
        """Identifica dependencias entre declaraciones"""
        dependencies = set()

        # Buscar referencias a otras secciones
        reference_patterns = [
            r'como\s+se\s+menciona\s+en',
            r'segÃºn\s+lo\s+establecido\s+en',
            r'de\s+acuerdo\s+con',
            r'en\s+lÃ-nea\s+con',
```

```
            r’siguiendo\s+lo\s+dispuesto’
        ]

        for pattern in reference_patterns:
            if re.search(pattern, sent.text, re.IGNORECASE):
                # Buscar la secciÃ³n referenciada
                for other_sent in doc.sents:
                    if other_sent != sent:
                        # Usar hash de los primeros 50 caracteres como ID
                        dependencies.add(other_sent.text[:50])

        return dependencies

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_context_window")
    def _get_context_window(self, text: str, start: int, end: int, window_size: int = 200) -> str:
        """Obtiene ventana de contexto alrededor de una posiciÃ³n"""
        context_start = max(0, start - window_size)
        context_end = min(len(text), end + window_size)
        return text[context_start:context_end]

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_similarity")
    def _calculate_similarity(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) -> float:
        """Calcula similaridad entre dos declaraciones"""
        if stmt_a.embedding is not None and stmt_b.embedding is not None:
            return float(1 - cosine(stmt_a.embedding, stmt_b.embedding))
        return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_similarity", "auto_param_L1310_15", 0.0)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction")
    def _classify_contradiction(self, text: str) -> float:
        """Clasifica probabilidad de contradicciÃ³n en texto"""
        try:
            result = self.contradiction_classifier(text)
            # Buscar score de contradicciÃ³n
            for item in result:
                if ’contradiction’ in item[’label’].lower():
                    return item[’score’]
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction", "auto_param_L1321_19", 0.0)
        except Exception as e:
            logger.warning(f"Error en clasificaciÃ³n de contradicciÃ³n: {e}")
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction", "auto_param_L1324_19", 0.0)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight")
    def _get_domain_weight(self, dimension: PolicyDimension) -> float:
        """Obtiene peso especÃ-fico del dominio"""
        weights = {
            PolicyDimension.DIAGNOSTICO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight", "auto_param_L1330_41", 0.8),
            PolicyDimension.ESTRATEGICO: 1.2,
            PolicyDimension.PROGRAMATICO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight", "auto_param_L1332_42", 1.0),
            PolicyDimension.FINANCIERO: 1.5,
            PolicyDimension.SEGUIMIENTO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight", "auto_param_L1334_41", 0.9),
```

```
            PolicyDimension.TERRITORIAL: 1.1
        }
        return weights.get(dimension, ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight", "auto_param_L1337_38", 1.0))

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._suggest_resolutions")
    def _suggest_resolutions(self, contradiction_type: ContradictionType) -> list[str]:
        """Sugiere resoluciones especÃficas por tipo de contradicciÃ³n"""
        suggestions = {
            ContradictionType.NUMERICAL_INCONSISTENCY: [
                "Verificar fuentes de datos y metodologÃas de cÃ¡lculo",
                "Establecer lÃnea base Ãºnica con validaciÃ³n tÃ©cnica",
                "Documentar supuestos y proyecciones utilizadas"
            ],
            ContradictionType.TEMPORAL_CONFLICT: [
                "Revisar cronograma maestro del plan",
                "Validar secuencia lÃ³gica de actividades",
                "Ajustar plazos segÃºn capacidad institucional"
            ],
            ContradictionType.SEMANTIC_OPPOSITION: [
                "Realizar taller de alineaciÃ³n conceptual",
                "Clarificar definiciones en glosario tÃ©cnico",
                "Priorizar objetivos segÃºn Plan Nacional de Desarrollo"
            ],
            ContradictionType.RESOURCE_ALLOCATION_MISMATCH: [
                "Realizar anÃ¡lisis de brechas financieras",
                "Priorizar inversiones segÃºn impacto social",
                "Explorar fuentes alternativas de financiaciÃ³n"
            ],
            ContradictionType.LOGICAL_INCOMPATIBILITY: [
                "Revisar cadena de valor de programas",
                "Validar teorÃa de cambio del plan",
                "Eliminar duplicidades y solapamientos"
            ]
        }
        return suggestions.get(contradiction_type, ["Revisar y ajustar segÃºn contexto"])

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._are_comparable_claims")
    def _are_comparable_claims(self, claim_a: dict, claim_b: dict) -> bool:
        """Determina si dos afirmaciones cuantitativas son comparables"""
        # Mismo tipo y contexto similar
        if claim_a['type'] != claim_b['type']:
            return False

        # Verificar si hablan del mismo concepto
        context_similarity = self._text_similarity(
            claim_a.get('context', ''),
            claim_b.get('context', '')
        )

        return context_similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._are_comparable_claims", "auto_param_L1384_36", 0.6)

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity")
    def _text_similarity(self, text_a: str, text_b: str) -> float:
        """Calcula similaridad simple entre textos"""
```

```
        if not text_a or not text_b:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1390_19", 0.0)

        # TokenizaciÃ³n simple
        tokens_a = set(text_a.lower().split())
        tokens_b = set(text_b.lower().split())

        if not tokens_a or not tokens_b:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1397_19", 0.0)

        # Coeficiente de Jaccard
        intersection = tokens_a & tokens_b
        union = tokens_a | tokens_b

        return len(intersection) / len(union) if union else ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1403_60", 0.0)

    def _calculate_numerical_divergence(
            self,
            claim_a: dict,
            claim_b: dict
    ) -> float | None:
        """Calcula divergencia entre valores numÃ©ricos"""
        value_a = claim_a.get('value', 0)
        value_b = claim_b.get('value', 0)

        if value_a == 0 and value_b == 0:
            return None

        # Divergencia relativa
        max_value = max(abs(value_a), abs(value_b))
        if max_value == 0:
            return None

        divergence = abs(value_a - value_b) / max_value
        return divergence

    def _statistical_significance_test(
            self,
            claim_a: dict,
            claim_b: dict
    ) -> float:
        """Realiza test de significancia estadÃstica"""
        value_a = claim_a.get('value', 0)
        value_b = claim_b.get('value', 0)

        # Test t de una muestra para diferencia significativa
        # Asumiendo distribuciÃ³n normal con varianza estimada
        diff = abs(value_a - value_b)
        pooled_value = (value_a + value_b) / 2

        if pooled_value == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1440_19", 1.0)  # No significativo
```

```

        # EstimaciÃ³n conservadora de error estÃ¡ndar
        se = pooled_value * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1443_28", 0.1)  # 10% de error estimado

        if se == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1446_19", 0.0)  # Altamente significativo

        # EstadÃstico t
        t_stat = diff / se

        # Valor p aproximado (two-tailed)
        p_value = 2 * (1 - stats.norm.cdf(abs(t_stat)))

        return p_value

    @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict")
    def _has_logical_conflict(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) -> bool:
        """Determina si hay conflicto lÃ³gico entre declaraciones"""
        # Verificar si las declaraciones tienen roles incompatibles
        if stmt_a.semantic_role and stmt_b.semantic_role:
            incompatible_roles = [
                ('objective', 'constraint'),
                ('strategy', 'constraint'),
                ('action', 'constraint')
            ]

            for role_pair in incompatible_roles:
                if (stmt_a.semantic_role in role_pair and
                        stmt_b.semantic_role in role_pair and
                        stmt_a.semantic_role != stmt_b.semantic_role):
                    return True

        # Verificar negaciÃ³n explÃcita
        negation_patterns = ['no', 'nunca', 'ningÃºn', 'sin', 'tampoco']
        has_negation_a = any(pattern in stmt_a.text.lower() for pattern in negation_patterns)
        has_negation_b = any(pattern in stmt_b.text.lower() for pattern in negation_patterns)

        # Si una tiene negaciÃ³n y otra no, y son similares, hay conflicto
        if has_negation_a != has_negation_b:
            similarity = self._calculate_similarity(stmt_a, stmt_b)
            if similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1481_28", 0.7):
                return True

        return False

    def _are_conflicting_allocations(
            self,
            amount_a: float,
            amount_b: float,
            total: float
    ) -> bool:
        """Determina si las asignaciones de recursos estÃ¡n en conflicto"""
        # Si la suma excede el total disponible
```

```
        if amount_a + amount_b > total * 1.1:  # 10% de margen
            return True

        # Si hay una diferencia muy grande entre asignaciones similares
        return abs(amount_a - amount_b) / max(amount_a, amount_b) > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1498_68", 0.5)

    def _determine_relation_type(
            self,
            stmt_a: PolicyStatement,
            stmt_b: PolicyStatement
    ) -> str:
        """Determina el tipo de relaciÃ³n entre dos declaraciones"""
        # Analizar roles semÃ¡nticos
        if stmt_a.semantic_role and stmt_b.semantic_role:
            if stmt_a.semantic_role == stmt_b.semantic_role:
                return "parallel"
            elif stmt_a.semantic_role in ["strategy", "objective"] and stmt_b.semantic_role == "action":
                return "enables"
            elif stmt_a.semantic_role == "action" and stmt_b.semantic_role in ["indicator", "resource"]:
                return "requires"

        # Analizar dependencias
        if stmt_a.dependencies & {stmt_b.text[:50]}:
            return "depends_on"

        # Por defecto, relaciÃ³n de similaridad
        return "related"

    def _calculate_severity(
            self,
            stmt_a: PolicyStatement,
            stmt_b: PolicyStatement
    ) -> float:
        """Calcula la severidad de una contradicciÃ³n entre declaraciones"""
        severity = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "severity", 0.5) # Refactored

        # Incrementar si las declaraciones estÃ¡n en la misma dimensiÃ³n
        if stmt_a.dimension == stmt_b.dimension:
            severity += ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1532_24", 0.2)

        # Incrementar si tienen muchas entidades en comÃºn
        common_entities = set(stmt_a.entities) & set(stmt_b.entities)
        if len(common_entities) > 0:
            severity += min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1537_28", 0.2), len(common_entities) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1537_56", 0.05))

        # Incrementar si tienen marcadores temporales en conflicto
        if stmt_a.temporal_markers and stmt_b.temporal_markers:
            severity += ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1541_24", 0.1)

        return min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1543
```

```
_19", 1.0), severity)



================================================================================
FILE: src/farfan_pipeline/analysis/derek_beach.py
================================================================================

#!/usr/bin/env python3
"""
Causal Deconstruction and Audit Framework (CDAF) v2.0
Framework de ProducciÃ³n para AnÃ¡lisis Causal de Planes de Desarrollo Territorial

THEORETICAL FOUNDATION (Derek Beach):
"A causal mechanism is a system of interlocking parts (entities engaging in
activities) that transmits causal forces from X to Y" (Beach 2016: 465)

This framework implements Theory-Testing Process Tracing with mechanistic evidence
evaluation using Beach’s evidential tests taxonomy (Beach & Pedersen 2019).

Author: AI Systems Architect
Version: 2.0.0 (Beach-Grounded Production Grade)
"""

import argparse
import hashlib
import json
import logging
import re
import sys
import warnings
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from pathlib import Path
from typing import (
    TYPE_CHECKING,
    Any,
    Literal,
    NamedTuple,
    TypedDict,
    cast,
)
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    import fitz

# Core dependencies
try:
    import networkx as nx
    import numpy as np
    import pandas as pd
    import spacy
    import yaml
    from fuzzywuzzy import fuzz, process
```

```
    from pydantic import BaseModel, Field, ValidationError, validator
    from pydot import Dot, Edge, Node
    from scipy.spatial.distance import cosine
    from scipy.special import rel_entr
except ImportError as e:
    print(f"ERROR: Dependencia faltante. Ejecute: pip install {e.name}")
    sys.exit(1)

# DNP Standards Integration
try:
    from dnp_integration import ValidadorDNP

    DNP_AVAILABLE = True
except ImportError:
    DNP_AVAILABLE = False
    warnings.warn("MÃ³dulos DNP no disponibles. ValidaciÃ³n DNP deshabilitada.", stacklevel=2)

# Refactored Bayesian Engine (F1.2: Architectural Refactoring)
try:
    from inference.bayesian_adapter import BayesianEngineAdapter

    REFACTORED_BAYESIAN_AVAILABLE = True
except ImportError:
    REFACTORED_BAYESIAN_AVAILABLE = False
    warnings.warn("Motor Bayesiano refactorizado no disponible. Usando implementaciÃ³n legacy.", stacklevel=2)

# Configure logging
logging.basicConfig(
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S'
)

# ============================================================================
# CONSTANTS
# ============================================================================
DEFAULT_CONFIG_FILE = "config.yaml"
EXTRACTION_REPORT_SUFFIX = "_extraction_confidence_report.json"
CAUSAL_MODEL_SUFFIX = "_causal_model.json"
DNP_REPORT_SUFFIX = "_dnp_compliance_report.txt"

# Type definitions
NodeType = Literal["programa", "producto", "resultado", "impacto"]
RigorStatus = Literal["fuerte", "dÃ©bil", "sin_evaluar"]
TestType = Literal["hoop_test", "smoking_gun", "doubly_decisive", "straw_in_wind"]
DynamicsType = Literal["suma", "decreciente", "constante", "indefinido"]

# ============================================================================
# BEACH THEORETICAL PRIMITIVES - Added to existing code
# ============================================================================

class BeachEvidentialTest:
    """
    Derek Beach evidential tests implementation (Beach & Pedersen 2019: Ch 5).

    FOUR-FOLD TYPOLOGY calibrated by necessity (N) and sufficiency (S):

```

```
    HOOP TEST [N: High, S: Low]:
    - Fail â\206\222 ELIMINATES hypothesis (definitive knock-out)
    - Pass â\206\222 Hypothesis survives but not proven
    - Example: "Responsible entity must be documented"

    SMOKING GUN [N: Low, S: High]:
    - Pass â\206\222 Strongly confirms hypothesis
    - Fail â\206\222 Doesn't eliminate (could be false negative)
    - Example: "Unique policy instrument only used for this mechanism"

    DOUBLY DECISIVE [N: High, S: High]:
    - Pass â\206\222 Conclusively confirms
    - Fail â\206\222 Conclusively eliminates
    - Extremely rare in social science

    STRAW-IN-WIND [N: Low, S: Low]:
    - Pass/Fail â\206\222 Marginal confidence change
    - Used for preliminary screening

    REFERENCE: Beach & Pedersen (2019), pp 117-126
    """

    @staticmethod
    def classify_test(necessity: float, sufficiency: float) -> TestType:
        """
        Classify evidential test type based on necessity and sufficiency.

        Beach calibration:
        - Necessity > 0.7 â\206\222 High necessity
        - Sufficiency > 0.7 â\206\222 High sufficiency
        """
        high_n = necessity > 0.7
        high_s = sufficiency > 0.7

        if high_n and high_s:
            return "doubly_decisive"
        elif high_n and not high_s:
            return "hoop_test"
        elif not high_n and high_s:
            return "smoking_gun"
        else:
            return "straw_in_wind"

    @staticmethod
    def apply_test_logic(test_type: TestType, evidence_found: bool,
                         prior: float, bayes_factor: float) -> tuple[float, str]:
        """
        Apply Beach test-specific logic to Bayesian updating.

        CRITICAL RULES:
        1. Hoop Test FAIL â\206\222 posterior â\211\210 0 (knock-out)
        2. Smoking Gun PASS â\206\222 multiply prior by large BF (>10)
        3. Doubly Decisive â\206\222 extreme updates (BF > 100 or < 0.01)

        Returns: (posterior_confidence, interpretation)
        """
```

```
        if test_type == "hoop_test":
            if not evidence_found:
                # KNOCK-OUT per Beach: "hypothesis must jump through hoop"
                return 0.01, "HOOP_TEST_FAILURE: Hypothesis eliminated"
            else:
                # Pass: necessary condition met, use standard Bayesian
                posterior = min(0.95, prior * bayes_factor)
                return posterior, "HOOP_TEST_PASSED: Hypothesis survives, not proven"

        elif test_type == "smoking_gun":
            if evidence_found:
                # Strong confirmation: unique evidence found
                posterior = min(0.98, prior * max(bayes_factor, 10.0))
                return posterior, "SMOKING_GUN_FOUND: Strong confirmation"
            else:
                # Doesn’t eliminate: could be false negative
                posterior = prior * 0.9  # slight penalty
                return posterior, "SMOKING_GUN_NOT_FOUND: Doesn’t eliminate"

        elif test_type == "doubly_decisive":
            if evidence_found:
                return 0.99, "DOUBLY_DECISIVE_CONFIRMED: Conclusive"
            else:
                return 0.01, "DOUBLY_DECISIVE_ELIMINATED: Conclusive"

        # Marginal update only
        elif evidence_found:
            posterior = min(0.95, prior * min(bayes_factor, 2.0))
            return posterior, "STRAW_IN_WIND: Weak support"
        else:
            posterior = max(0.05, prior / min(bayes_factor, 2.0))
            return posterior, "STRAW_IN_WIND: Weak disconfirmation"

# ============================================================================
# Custom Exceptions - Structured Error Semantics
# ============================================================================

class CDAFException(Exception):
    """Base exception for CDAF framework with structured payloads"""

    def __init__(self, message: str, details: dict[str, Any] | None = None,
                 stage: str | None = None, recoverable: bool = False) -> None:
        self.message = message
        self.details = details or {}
        self.stage = stage
        self.recoverable = recoverable
        super().__init__(self._format_message())

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFException._format_message")
    def _format_message(self) -> str:
        """Format error message with structured information"""
        parts = ["[CDAF Error]"]
        if self.stage:
            parts.append(f"[Stage: {self.stage}]")
        parts.append(self.message)
        if self.details:
```

```
            parts.append(f"Details: {json.dumps(self.details, indent=2)}")
        return " ".join(parts)

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFException.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert exception to structured dictionary"""
        return {
            'error_type': self.__class__.__name__,
            'message': self.message,
            'details': self.details,
            'stage': self.stage,
            'recoverable': self.recoverable
        }

class CDAFValidationError(CDAFException):
    """Configuration or data validation error"""
    pass

class CDAFProcessingError(CDAFException):
    """Error during document processing"""
    pass

class CDAFBayesianError(CDAFException):
    """Error during Bayesian inference"""
    pass

class CDAFConfigError(CDAFException):
    """Configuration loading or validation error"""
    pass

# ============================================================================
# Pydantic Configuration Models - Schema Validation at Load Time
# ============================================================================

class BayesianThresholdsConfig(BaseModel):
    """Bayesian inference thresholds configuration"""
    kl_divergence: float = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "kl_divergence", 0.01),
        ge=0.0,
        le=1.0,
        description="KL divergence threshold for convergence"
    )
    convergence_min_evidence: int = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "convergence_min_evidence", 2),
        ge=1,
        description="Minimum evidence count for convergence check"
    )
    prior_alpha: float = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "prior_alpha", 2.0),
        ge=0.1,
        description="Default alpha parameter for Beta prior"
    )
    prior_beta: float = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "prior_beta", 2.0),
        ge=0.1,
        description="Default beta parameter for Beta prior"
```

```
    )
    laplace_smoothing: float = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "laplace_smoothing", 1.0),
        ge=0.0,
        description="Laplace smoothing parameter"
    )

class MechanismTypeConfig(BaseModel):
    """Mechanism type prior probabilities"""
    administrativo: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "administrativo", 0.30), ge=0.0, le=1.0)
    tecnico: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "tecnico", 0.25), ge=0.0, le=1.0)
    financiero: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "financiero", 0.20), ge=0.0, le=1.0)
    politico: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "politico", 0.15), ge=0.0, le=1.0)
    mixto: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "mixto", 0.10), ge=0.0, le=1.0)

    @validator(’*’, pre=True, always=True)
    def check_sum_to_one(cls, v, values):
        """Validate that probabilities sum to approximately 1.0"""
        if len(values) == 4:  # All fields loaded
            total = sum(values.values()) + v
            if abs(total - 1.0) > 0.01:
                raise ValueError(f"Mechanism type priors must sum to 1.0, got {total}")
        return v

class PerformanceConfig(BaseModel):
    """Performance and optimization settings"""
    enable_vectorized_ops: bool = Field(
        default=True,
        description="Use vectorized numpy operations where possible"
    )
    enable_async_processing: bool = Field(
        default=False,
        description="Enable async processing for large PDFs (experimental)"
    )
    max_context_length: int = Field(
        default=1000,
        ge=100,
        description="Maximum context length for spaCy processing"
    )
    cache_embeddings: bool = Field(
        default=True,
        description="Cache spaCy embeddings for reuse"
    )

class SelfReflectionConfig(BaseModel):
    """Self-reflective learning configuration"""
    enable_prior_learning: bool = Field(
        default=False,
        description="Enable learning from audit feedback to update priors"
    )
    feedback_weight: float = Field(
        default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.SelfReflectionConfig", "feedback_weight", 0.1),
        ge=0.0,
        le=1.0,
        description="Weight for feedback in prior updates (0=ignore, 1=full)"
```

```
    )
    prior_history_path: str | None = Field(
        default=None,
        description="Path to save/load historical priors"
    )
    min_documents_for_learning: int = Field(
        default=5,
        ge=1,
        description="Minimum documents before applying learned priors"
    )

class CDAFConfigSchema(BaseModel):
    """Complete CDAF configuration schema with validation"""
    patterns: dict[str, str] = Field(
        description="Regex patterns for document parsing"
    )
    lexicons: dict[str, Any] = Field(
        description="Lexicons for causal logic, classification, etc."
    )
    entity_aliases: dict[str, str] = Field(
        description="Entity name aliases and mappings"
    )
    verb_sequences: dict[str, int] = Field(
        description="Verb sequence ordering for temporal coherence"
    )
    bayesian_thresholds: BayesianThresholdsConfig = Field(
        default_factory=BayesianThresholdsConfig,
        description="Bayesian inference thresholds"
    )
    mechanism_type_priors: MechanismTypeConfig = Field(
        default_factory=MechanismTypeConfig,
        description="Prior probabilities for mechanism types"
    )
    performance: PerformanceConfig = Field(
        default_factory=PerformanceConfig,
        description="Performance and optimization settings"
    )
    self_reflection: SelfReflectionConfig = Field(
        default_factory=SelfReflectionConfig,
        description="Self-reflective learning configuration"
    )

    class Config:
        extra = 'allow'  # Allow additional fields for extensibility

class GoalClassification(NamedTuple):
    """Classification structure for goals"""
    type: NodeType
    dynamics: DynamicsType
    test_type: TestType
    confidence: float

class EntityActivity(NamedTuple):
    """
    Entity-Activity tuple for mechanism parts (Beach 2016).

```

```
    BEACH DEFINITION:
    "A mechanism part consists of an entity (organization, actor, structure)
    engaging in an activity that transmits causal forces" (Beach 2016: 465)

    This is the FUNDAMENTAL UNIT of mechanistic evidence in Process Tracing.
    """
    entity: str
    activity: str
    verb_lemma: str
    confidence: float

class CausalLink(TypedDict):
    """Structure for causal links in the graph"""
    source: str
    target: str
    logic: str
    strength: float
    evidence: list[str]
    posterior_mean: float | None
    posterior_std: float | None
    kl_divergence: float | None
    converged: bool | None

class AuditResult(TypedDict):
    """Audit result structure"""
    passed: bool
    warnings: list[str]
    errors: list[str]
    recommendations: list[str]

@dataclass
class MetaNode:
    """Comprehensive node structure for goals/metas"""
    id: str
    text: str
    type: NodeType
    baseline: float | str | None = None
    target: float | str | None = None
    unit: str | None = None
    responsible_entity: str | None = None
    entity_activity: EntityActivity | None = None
    financial_allocation: float | None = None
    unit_cost: float | None = None
    rigor_status: RigorStatus = "sin_evaluar"
    dynamics: DynamicsType = "indefinido"
    test_type: TestType = "straw_in_wind"
    contextual_risks: list[str] = field(default_factory=list)
    causal_justification: list[str] = field(default_factory=list)
    audit_flags: list[str] = field(default_factory=list)
    confidence_score: float = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MetaNode", "confidence_score", 0.0)

class ConfigLoader:
    """External configuration management with Pydantic schema validation"""

    def __init__(self, config_path: Path) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
```

```
        self.config_path = config_path
        self.config: dict[str, Any] = {}
        self.validated_config: CDAFConfigSchema | None = None
        # HARMONIC FRONT 4: Track uncertainty over iterations
        self._uncertainty_history: list[float] = []
        self._load_config()
        self._validate_config()
        self._load_uncertainty_history()

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_config")
    def _load_config(self) -> None:
        """Load YAML configuration file"""
        try:
            with open(self.config_path, 'r', encoding='utf-8') as f:
                self.config = yaml.safe_load(f)
            self.logger.info(f"ConfiguraciÃ³n cargada desde {self.config_path}")
        except FileNotFoundError:
            self.logger.warning(f"Archivo de configuraciÃ³n no encontrado: {self.config_path}")
            self._load_default_config()
        except Exception as e:
            raise CDAFConfigError(
                "Error cargando configuraciÃ³n",
                details={'path': str(self.config_path), 'error': str(e)},
                stage="config_load",
                recoverable=True
            )

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config")
    def _load_default_config(self) -> None:
        """Load default configuration if custom fails"""
        self.config = {
            'patterns': {
                'section_titles': r'^(?:CAPÃ\215TULO|ARTÃ\215CULO|PARTE)\s+[\dIVX]+',
                'goal_codes': r'[MP][RIP]-\d{3}',
                'numeric_formats': r'[\d,]+(?:\.\d+)?%?',
                'table_headers': r'(?:PROGRAMA|META|INDICADOR|LÃ\215NEA BASE|VALOR ESPERADO)',
                'financial_headers': r'(?:PRESUPUESTO|VALOR|MONTO|INVERSIÃ\223N)'
            },
            'lexicons': {
                'causal_logic': [
                    'gracias a', 'con el fin de', 'para lograr', 'mediante',
                    'a travÃ©s de', 'como resultado de', 'debido a', 'porque',
                    'por medio de', 'permitirÃ¡', 'contribuirÃ¡ a'
                ],
                'goal_classification': {
                    'tasa': 'decreciente',
                    'Ã-ndice': 'constante',
                    'nÃºmero': 'suma',
                    'porcentaje': 'constante',
                    'cantidad': 'suma',
                    'cobertura': 'suma'
                },
                'contextual_factors': [
                    'riesgo', 'amenaza', 'obstÃ¡culo', 'limitaciÃ³n',
                    'restricciÃ³n', 'desafÃ-o', 'brecha', 'dÃ©ficit',
                    'vulnerabilidad', 'hipÃ³tesis alternativa'
```

```
                ],
                'administrative_keywords': [
                    'gestiÃ³n', 'administraciÃ³n', 'coordinaciÃ³n', 'regulaciÃ³n',
                    'normativa', 'institucional', 'gobernanza', 'reglamento',
                    'decreto', 'resoluciÃ³n', 'acuerdo'
                ]
            },
            'entity_aliases': {
                'SEC GOB': 'SecretarÃa de Gobierno',
                'SEC PLAN': 'SecretarÃa de PlaneaciÃ³n',
                'SEC HAC': 'SecretarÃa de Hacienda',
                'SEC SALUD': 'SecretarÃa de Salud',
                'SEC EDU': 'SecretarÃa de EducaciÃ³n',
                'SEC INFRA': 'SecretarÃa de Infraestructura'
            },
            'verb_sequences': {
                'diagnosticar': 1,
                'identificar': 2,
                'analizar': 3,
                'diseÃ±ar': 4,
                'planificar': 5,
                'implementar': 6,
                'ejecutar': 7,
                'monitorear': 8,
                'evaluar': 9
            },
            # Bayesian thresholds - now externalized
            'bayesian_thresholds': {
                'kl_divergence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "kl_divergence", 0.01),
                'convergence_min_evidence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "convergence_min_evidence", 2),
                'prior_alpha': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "prior_alpha", 2.0),
                'prior_beta': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "prior_beta", 2.0),
                'laplace_smoothing': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "laplace_smoothing", 1.0)
            },
            # Mechanism type priors - now externalized
            'mechanism_type_priors': {
                'administrativo': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "administrativo", 0.30),
                'tecnico': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "tecnico", 0.25),
                'financiero': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "financiero", 0.20),
                'politico': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "politico", 0.15),
                'mixto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "mixto", 0.10)
            },
            # Performance settings
            'performance': {
                'enable_vectorized_ops': True,
                'enable_async_processing': False,
                'max_context_length': 1000,
                'cache_embeddings': True
            },
            # Self-reflection settings
            'self_reflection': {
                'enable_prior_learning': False,
                'feedback_weight': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "feedback_weight", 0.1),
                'prior_history_path': None,
                'min_documents_for_learning': 5
```

```
            }
        }
        self.logger.warning("Usando configuraciÃ³n por defecto")

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._validate_config")
    def _validate_config(self) -> None:
        """Validate configuration structure using Pydantic schema"""
        try:
            # Validate with Pydantic schema
            self.validated_config = CDAFConfigSchema(**self.config)
            self.logger.info("â\234\223 ConfiguraciÃ³n validada exitosamente con esquema Pydantic")
        except ValidationError as e:
            error_details = {
                ’validation_errors’: [
                    {
                        ’field’: ’.’.join(str(x) for x in err[’loc’]),
                        ’error’: err[’msg’],
                        ’type’: err[’type’]
                    }
                    for err in e.errors()
                ]
            }
            raise CDAFValidationError(
                "ConfiguraciÃ³n invÃ¡lida - errores de esquema",
                details=error_details,
                stage="config_validation",
                recoverable=False
            )

        # Legacy validation for required sections
        required_sections = [’patterns’, ’lexicons’, ’entity_aliases’, ’verb_sequences’]
        for section in required_sections:
            if section not in self.config:
                self.logger.warning(f"SecciÃ³n faltante en configuraciÃ³n: {section}")
                self.config[section] = {}

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get")
    def get(self, key: str, default: Any = None) -> Any:
        """Get configuration value with dot notation support"""
        keys = key.split(’.’)
        value = self.config
        for k in keys:
            if isinstance(value, dict):
                value = value.get(k, default)
            else:
                return default
        return value

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_bayesian_threshold")
    def get_bayesian_threshold(self, key: str) -> float:
        """Get Bayesian threshold with type safety"""
        if self.validated_config:
            return getattr(self.validated_config.bayesian_thresholds, key)
        return self.get(f’bayesian_thresholds.{key}’, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_bayesian_threshold", "default", 0.01))

```

```
    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior")
    def get_mechanism_prior(self, mechanism_type: str) -> float:
        """Get mechanism type prior probability with type safety"""
        if self.validated_config:
            return getattr(self.validated_config.mechanism_type_priors, mechanism_type, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior", "default", 0.0))
        return self.get(f'mechanism_type_priors.{mechanism_type}', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior", "default", 0.0))

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_performance_setting")
    def get_performance_setting(self, key: str) -> Any:
        """Get performance setting with type safety"""
        if self.validated_config:
            return getattr(self.validated_config.performance, key)
        return self.get(f'performance.{key}')

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback")
    def update_priors_from_feedback(self, feedback_data: dict[str, Any]) -> None:
        """
        Self-reflective loop: Update priors based on audit feedback
        Implements frontier paradigm of learning from results

        HARMONIC FRONT 4 ENHANCEMENT:
        - Applies penalties to mechanism types with implementation_failure flags
        - Heavily penalizes "miracle" mechanisms failing necessity/sufficiency tests
        - Ensures mean mech_uncertainty decreases by â\211¥5% over iterations
        """
        if not self.validated_config or not self.validated_config.self_reflection.enable_prior_learning:
            self.logger.debug("Prior learning disabled")
            return

        feedback_weight = self.validated_config.self_reflection.feedback_weight

        # Track initial priors for uncertainty measurement
        initial_priors = {}
        for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
            if hasattr(self.validated_config.mechanism_type_priors, attr):
                initial_priors[attr] = getattr(self.validated_config.mechanism_type_priors, attr)

        # Update mechanism type priors based on observed frequencies
        if 'mechanism_frequencies' in feedback_data:
            for mech_type, observed_freq in feedback_data['mechanism_frequencies'].items():
                if hasattr(self.validated_config.mechanism_type_priors, mech_type):
                    current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
                    # Weighted update: new_prior = (1-weight)*current + weight*observed
                    updated_prior = (1 - feedback_weight) * current_prior + feedback_weight * observed_freq
                    setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
                    self.config['mechanism_type_priors'][mech_type] = updated_prior

        # NEW: Apply penalty factors for failing mechanism types
        if 'penalty_factors' in feedback_data:
            penalty_weight = feedback_weight * 1.5  # Heavier penalty than positive feedback
            for mech_type, penalty_factor in feedback_data['penalty_factors'].items():
                if hasattr(self.validated_config.mechanism_type_priors, mech_type):
                    current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
                    # Apply penalty: reduce prior for frequently failing types
```

```
                    penalized_prior = current_prior * penalty_factor
                    # Blend with current
                    updated_prior = (1 - penalty_weight) * current_prior + penalty_weight * penalized_prior
                    setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
                    self.config['mechanism_type_priors'][mech_type] = updated_prior
                    self.logger.info(f"Applied penalty to {mech_type}: {current_prior:.4f} -> {updated_prior:.4f}")

        # NEW: Heavy penalty for "miracle" mechanisms failing necessity/sufficiency
        test_failures = feedback_data.get('test_failures', {})
        if test_failures.get('necessity_failures', 0) > 0 or test_failures.get('sufficiency_failures', 0) > 0:
            # If failures exist, apply additional penalty to 'politico' (often "miracle" type)
            # and 'mixto' (vague mechanism types)
            miracle_types = ['politico', 'mixto']
            miracle_penalty = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback", "miracle_penalty", 0.85) # Refactored
            for mech_type in miracle_types:
                if hasattr(self.validated_config.mechanism_type_priors, mech_type):
                    current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
                    updated_prior = current_prior * miracle_penalty
                    setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
                    self.config['mechanism_type_priors'][mech_type] = updated_prior
                    self.logger.info(
                        f"Miracle mechanism penalty for {mech_type}: {current_prior:.4f} -> {updated_prior:.4f}")

        # Renormalize to ensure priors sum to ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback", "auto_param_L686_46", 1.0)
        total_prior = sum(
            getattr(self.validated_config.mechanism_type_priors, attr)
            for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']
            if hasattr(self.validated_config.mechanism_type_priors, attr)
        )

        if total_prior > 0:
            for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
                if hasattr(self.validated_config.mechanism_type_priors, attr):
                    current = getattr(self.validated_config.mechanism_type_priors, attr)
                    normalized = current / total_prior
                    setattr(self.validated_config.mechanism_type_priors, attr, normalized)
                    self.config['mechanism_type_priors'][attr] = normalized

        # Calculate uncertainty reduction for quality criteria
        final_priors = {}
        for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
            if hasattr(self.validated_config.mechanism_type_priors, attr):
                final_priors[attr] = getattr(self.validated_config.mechanism_type_priors, attr)

        # Calculate entropy as uncertainty measure
        initial_entropy = -sum(p * np.log(p + 1e-10) for p in initial_priors.values() if p > 0)
        final_entropy = -sum(p * np.log(p + 1e-10) for p in final_priors.values() if p > 0)
        uncertainty_reduction = ((initial_entropy - final_entropy) / max(initial_entropy, 1e-10)) * 100

        self.logger.info(f"Uncertainty reduction: {uncertainty_reduction:.2f}%")

        # Save updated priors if history path configured
        if self.validated_config.self_reflection.prior_history_path:
            self._save_prior_history(feedback_data, uncertainty_reduction)
```

```
        self.logger.info(f"Priors actualizados con peso de retroalimentaciÃ³n {feedback_weight}")

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._save_prior_history")
    def _save_prior_history(self, feedback_data: dict[str, Any] | None = None,
                            uncertainty_reduction: float | None = None) -> None:
        """
        Save prior history for learning across documents

        HARMONIC FRONT 4 ENHANCEMENT:
        - Tracks uncertainty reduction over iterations
        - Records penalty applications and test failures
        """
        if not self.validated_config or not self.validated_config.self_reflection.prior_history_path:
            return

        try:
            history_path = Path(self.validated_config.self_reflection.prior_history_path)
            history_path.parent.mkdir(parents=True, exist_ok=True)

            # Load existing history if available
            history_records = []
            if history_path.exists():
                try:
                    with open(history_path, 'r', encoding='utf-8') as f:
                        existing_data = json.load(f)
                    if isinstance(existing_data, list):
                        history_records = existing_data
                    elif isinstance(existing_data, dict) and 'history' in existing_data:
                        history_records = existing_data['history']
                except json.JSONDecodeError:
                    self.logger.warning("Existing history file corrupted, starting fresh")

            # Create new record
            history_record = {
                'mechanism_type_priors': dict(self.config.get('mechanism_type_priors', {})),
                'timestamp': pd.Timestamp.now().isoformat(),
                'version': '2.0'
            }

            # Add feedback metrics if available
            if feedback_data:
                history_record['audit_quality'] = feedback_data.get('audit_quality', {})
                history_record['test_failures'] = feedback_data.get('test_failures', {})
                history_record['penalty_factors'] = feedback_data.get('penalty_factors', {})

            if uncertainty_reduction is not None:
                history_record['uncertainty_reduction_percent'] = uncertainty_reduction

            history_records.append(history_record)

            # Save complete history
            history_data = {
                'version': '2.0',
                'harmonic_front': 4,
                'last_updated': pd.Timestamp.now().isoformat(),
```

```
                'total_iterations': len(history_records),
                'history': history_records
            }

            with open(history_path, 'w', encoding='utf-8') as f:
                json.dump(history_data, f, indent=2)

            self.logger.info(f"Historial de priors guardado en {history_path} (iteraciÃ³n {len(history_records)})")
        except Exception as e:
            self.logger.warning(f"Error guardando historial de priors: {e}")

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_uncertainty_history")
    def _load_uncertainty_history(self) -> None:
        """
        Load historical uncertainty measurements

        HARMONIC FRONT 4: Required for tracking â\211¥5% reduction over 10 iterations
        """
        if not self.validated_config or not self.validated_config.self_reflection.prior_history_path:
            return

        try:
            history_path = Path(self.validated_config.self_reflection.prior_history_path)
            if history_path.exists():
                with open(history_path, 'r', encoding='utf-8') as f:
                    history_data = json.load(f)
                if isinstance(history_data, dict) and 'history' in history_data:
                    # Extract uncertainty from each record
                    for record in history_data['history']:
                        if 'uncertainty_reduction_percent' in record:
                            self._uncertainty_history.append(
                                record['uncertainty_reduction_percent']
                            )
                self.logger.info(f"Loaded {len(self._uncertainty_history)} uncertainty measurements")
        except Exception as e:
            self.logger.warning(f"Could not load uncertainty history: {e}")

    @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.check_uncertainty_reduction_criterion")
    def check_uncertainty_reduction_criterion(self, current_uncertainty: float) -> dict[str, Any]:
        """
        Check if mean mechanism_type uncertainty has decreased â\211¥5% over 10 iterations

        HARMONIC FRONT 4 QUALITY CRITERIA:
        Success verified if mean mech_uncertainty decreases by â\211¥5% over 10 sequential PDM analyses
        """
        self._uncertainty_history.append(current_uncertainty)

        # Keep only last 10 iterations
        recent_history = self._uncertainty_history[-10:]

        result = {
            'current_uncertainty': current_uncertainty,
            'iterations_tracked': len(recent_history),
            'criterion_met': False,
            'reduction_percent': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.check_uncertainty_reduction_criterion", "reduction_percent", 0.0),
```

```
            ’status’: ’insufficient_data’
        }

        if len(recent_history) >= 10:
            initial_uncertainty = recent_history[0]
            final_uncertainty = recent_history[-1]

            if initial_uncertainty > 0:
                reduction_percent = ((initial_uncertainty - final_uncertainty) / initial_uncertainty) * 100
                result[’reduction_percent’] = reduction_percent
                result[’criterion_met’] = reduction_percent >= 5.0
                result[’status’] = ’success’ if result[’criterion_met’] else ’needs_improvement’

                self.logger.info(
                    f"Uncertainty reduction over 10 iterations: {reduction_percent:.2f}% "
                    f"(criterion: â\211¥5%, met: {result[’criterion_met’]})"
                )
        else:
            self.logger.info(
                f"Uncertainty tracking: {len(recent_history)}/10 iterations "
                f"(need {10 - len(recent_history)} more for criterion check)"
            )

        return result

class PDFProcessor:
    """Advanced PDF processing and extraction"""

    def __init__(self, config: ConfigLoader, retry_handler=None) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.document: fitz.Document | None = None
        self.text_content: str = ""
        self.tables: list[pd.DataFrame] = []
        self.metadata: dict[str, Any] = {}
        self.retry_handler = retry_handler

    @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.load_document")
    def load_document(self, pdf_path: Path) -> bool:
        """Load PDF document with retry logic"""
        if self.retry_handler:
            try:
                from farfan_pipeline.analysis.retry_handler import DependencyType

                @self.retry_handler.with_retry(
                    DependencyType.PDF_PARSER,
                    operation_name="open_pdf",
                    exceptions=(IOError, OSError, RuntimeError)
                )
                def load_with_retry():
                    import fitz
                    doc = fitz.open(pdf_path)
                    self.logger.info(f"PDF cargado: {pdf_path.name} ({len(doc)} pÃ¡ginas)")
                    return doc

                self.document = load_with_retry()
```

```
                self.metadata = self.document.metadata
                return True
            except Exception as e:
                self.logger.error(f"Error cargando PDF: {e}")
                return False
        else:
            # Fallback without retry
            try:
                import fitz
                self.document = fitz.open(pdf_path)
                self.metadata = self.document.metadata
                self.logger.info(f"PDF cargado: {pdf_path.name} ({len(self.document)} pÃ¡ginas)")
                return True
            except Exception as e:
                self.logger.error(f"Error cargando PDF: {e}")
                return False

    @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.extract_text")
    def extract_text(self) -> str:
        """Extract all text from PDF"""
        if not self.document:
            return ""

        text_parts = []
        for page_num, page in enumerate(self.document, 1):
            try:
                text = page.get_text()
                text_parts.append(text)
                self.logger.debug(f"Texto extraÃ-do de pÃ¡gina {page_num}")
            except Exception as e:
                self.logger.warning(f"Error extrayendo texto de pÃ¡gina {page_num}: {e}")

        self.text_content = "\n".join(text_parts)
        self.logger.info(f"Texto total extraÃ-do: {len(self.text_content)} caracteres")
        return self.text_content

    @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.extract_tables")
    def extract_tables(self) -> list[pd.DataFrame]:
        """Extract tables from PDF"""
        if not self.document:
            return []

        table_pattern = re.compile(
            self.config.get('patterns.table_headers', r'PROGRAMA|META|INDICADOR'),
            re.IGNORECASE
        )

        for page_num, page in enumerate(self.document, 1):
            try:
                tabs = page.find_tables()
                if tabs:
                    for tab in tabs:
                        try:
                            df = pd.DataFrame(tab.extract())
                            if not df.empty and len(df.columns) > 1:
                                # Check if this is a relevant table
```

```
                                header_text = ' '.join(str(cell) for cell in df.iloc[0] if cell)
                                if table_pattern.search(header_text):
                                    self.tables.append(df)
                                    self.logger.info(f"Tabla extraÃda de pÃ¡gina {page_num}: {df.shape}")
                        except Exception as e:
                            self.logger.warning(f"Error procesando tabla en pÃ¡gina {page_num}: {e}")
            except Exception as e:
                self.logger.debug(f"Error extrayendo tablas de pÃ¡gina {page_num}: {e}")

        self.logger.info(f"Total de tablas extraÃdas: {len(self.tables)}")
        return self.tables

    @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.extract_sections")
    def extract_sections(self) -> dict[str, str]:
        """Extract document sections based on patterns"""
        sections = {}
        section_pattern = re.compile(
            self.config.get('patterns.section_titles', r'^(?:CAPÃ\215TULO|ARTÃ\215CULO)\s+[\dIVX]+'),
            re.MULTILINE | re.IGNORECASE
        )

        matches = list(section_pattern.finditer(self.text_content))

        for i, match in enumerate(matches):
            section_title = match.group().strip()
            start_pos = match.end()
            end_pos = matches[i + 1].start() if i + 1 < len(matches) else len(self.text_content)
            sections[section_title] = self.text_content[start_pos:end_pos].strip()

        self.logger.info(f"Secciones identificadas: {len(sections)}")
        return sections

class CausalExtractor:
    """Extract and structure causal chains from text"""

    def __init__(self, config: ConfigLoader, nlp_model: spacy.Language) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.nlp = nlp_model
        self.graph = nx.DiGraph()
        self.nodes: dict[str, MetaNode] = {}
        self.causal_chains: list[CausalLink] = []

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor.extract_causal_hierarchy")
    def extract_causal_hierarchy(self, text: str) -> nx.DiGraph:
        """Extract complete causal hierarchy from text"""
        # Extract goals/metas
        goals = self._extract_goals(text)

        # Build hierarchy
        for goal in goals:
            self._add_node_to_graph(goal)

        # Extract causal connections
        self._extract_causal_links(text)

```

```
        # Build hierarchy based on goal types
        self._build_type_hierarchy()

        self.logger.info(f"Grafo causal construido: {self.graph.number_of_nodes()} nodos, "
                         f"{self.graph.number_of_edges()} aristas")
        return self.graph

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_goals")
    def _extract_goals(self, text: str) -> list[MetaNode]:
        """Extract all goals from text"""
        goals = []
        goal_pattern = re.compile(
            self.config.get('patterns.goal_codes', r'[MP][RIP]-\d{3}'),
            re.IGNORECASE
        )

        for match in goal_pattern.finditer(text):
            goal_id = match.group().upper()
            context_start = max(0, match.start() - 500)
            context_end = min(len(text), match.end() + 500)
            context = text[context_start:context_end]

            goal = self._parse_goal_context(goal_id, context)
            if goal:
                goals.append(goal)
                self.nodes[goal.id] = goal

        self.logger.info(f"Metas extraÃ-das: {len(goals)}")
        return goals

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._parse_goal_context")
    def _parse_goal_context(self, goal_id: str, context: str) -> MetaNode | None:
        """Parse goal context to extract structured information"""
        # Determine goal type
        if goal_id.startswith('MP'):
            node_type = 'producto'
        elif goal_id.startswith('MR'):
            node_type = 'resultado'
        elif goal_id.startswith('MI'):
            node_type = 'impacto'
        else:
            node_type = 'programa'

        # Extract numerical values
        numeric_pattern = re.compile(
            self.config.get('patterns.numeric_formats', r'[\d,]+(?:\.\d+)?%?')
        )
        numbers = numeric_pattern.findall(context)

        # Process with spaCy
        doc = self.nlp(context[:1000])

        # Extract entities
        entities = [ent.text for ent in doc.ents if ent.label_ in ['ORG', 'PER', 'LOC']]

        # Create goal node
```

```
        goal = MetaNode(
            id=goal_id,
            text=context[:200].strip(),
            type=cast("NodeType", node_type),
            baseline=numbers[0] if len(numbers) > 0 else None,
            target=numbers[1] if len(numbers) > 1 else None,
            responsible_entity=entities[0] if entities else None
        )

        return goal

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_goal_text")
    def _extract_goal_text(self, text: str, **kwargs) -> str | None:
        """
        Extract the text content associated with a specific goal ID.

        This method extracts goal text from the provided document text. It can work
        in two modes:
        1. If a goal_id is provided in kwargs, it extracts text for that specific goal
        2. Otherwise, it returns the first goal text found in the document

        Args:
            text: The full document text
            **kwargs: Additional parameters including optional 'goal_id', 'data',
                     'sentences', 'tables'

        Returns:
            The extracted text for the goal, or None if not found
        """
        # Get goal_id from kwargs if provided, otherwise look for data parameter
        goal_id = kwargs.get('goal_id')
        kwargs.get('data')

        # If no goal_id specified, try to extract the first goal from text
        if not goal_id:
            goal_pattern = re.compile(
                r'\b[MP][RIP]-\d{3}\b',
                re.IGNORECASE
            )
            match = goal_pattern.search(text)
            if match:
                goal_id = match.group().upper()
            else:
                # No goal found in text
                return None

        # Now extract the context around the goal_id
        goal_pattern = re.compile(
            rf'\b{re.escape(goal_id)}\b',
            re.IGNORECASE
        )

        match = goal_pattern.search(text)
        if not match:
            return None

```

```
        # Extract context around the goal ID
        context_start = max(0, match.start() - 500)
        context_end = min(len(text), match.end() + 500)
        context = text[context_start:context_end]

        return context.strip()

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._add_node_to_graph")
    def _add_node_to_graph(self, node: MetaNode) -> None:
        """Add node to causal graph"""
        node_dict = asdict(node)
        # Convert NamedTuple to dict for JSON serialization
        if node.entity_activity:
            node_dict['entity_activity'] = node.entity_activity._asdict()
        self.graph.add_node(node.id, **node_dict)

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links")
    def _extract_causal_links(self, text: str) -> None:
        """
        AGUJA I: El Prior Informado Adaptativo
        Extract causal links using Bayesian inference with adaptive priors
        """
        causal_keywords = self.config.get('lexicons.causal_logic', [])

        # Get externalized thresholds from configuration
        kl_threshold = self.config.get_bayesian_threshold('kl_divergence')
        convergence_min_evidence = self.config.get_bayesian_threshold('convergence_min_evidence')

        # Track evidence for each potential link
        link_evidence: dict[tuple[str, str], list[dict[str, Any]]] = defaultdict(list)

        # Phase 1: Collect all evidence
        for keyword in causal_keywords:
            pattern = re.compile(
                rf'({"|".join(re.escape(nid) for nid in self.nodes)})'
                rf'\s+{re.escape(keyword)}\s+'
                rf'({"|".join(re.escape(nid) for nid in self.nodes)})',
                re.IGNORECASE
            )

            for match in pattern.finditer(text):
                source = match.group(1).upper()
                target = match.group(2).upper()
                logic = match.group(0)

                if source in self.nodes and target in self.nodes:
                    # Extract context around the match for language specificity analysis
                    context_start = max(0, match.start() - 100)
                    context_end = min(len(text), match.end() + 100)
                    match_context = text[context_start:context_end]

                    # Calculate evidence components
                    evidence = {
                        'keyword': keyword,
                        'logic': logic,
                        'match_position': match.start(),
```

```
                        'semantic_distance': self._calculate_semantic_distance(source, target),
                        'type_transition_prior': self._calculate_type_transition_prior(source, target),
                        'language_specificity': self._calculate_language_specificity(keyword, None, match_context),
                        'temporal_coherence': self._assess_temporal_coherence(source, target),
                        'financial_consistency': self._assess_financial_consistency(source, target),
                        'textual_proximity': self._calculate_textual_proximity(source, target, text)
                    }

                    link_evidence[(source, target)].append(evidence)

        # Phase 2: Bayesian inference for each link
        for (source, target), evidences in link_evidence.items():
            # Initialize prior distribution
            prior_mean, prior_alpha, prior_beta = self._initialize_prior(source, target)

            # Incremental Bayesian update
            posterior_alpha = prior_alpha
            posterior_beta = prior_beta
            kl_divs = []

            for evidence in evidences:
                # Calculate likelihood components
                likelihood = self._calculate_composite_likelihood(evidence)

                # Update Beta distribution parameters
                # Using Beta-Binomial conjugate prior
                posterior_alpha += likelihood
                posterior_beta += (1 - likelihood)

                # Calculate KL divergence for convergence check
                if len(kl_divs) > 0:
                    prior_dist = np.array([posterior_alpha - likelihood, posterior_beta - (1 - likelihood)])
                    prior_dist = prior_dist / prior_dist.sum()
                    posterior_dist = np.array([posterior_alpha, posterior_beta])
                    posterior_dist = posterior_dist / posterior_dist.sum()
                    kl_div = float(np.sum(rel_entr(posterior_dist, prior_dist)))
                    kl_divs.append(kl_div)

            # Calculate posterior statistics
            posterior_mean = posterior_alpha / (posterior_alpha + posterior_beta)
            posterior_var = (posterior_alpha * posterior_beta) / (
                    (posterior_alpha + posterior_beta) ** 2 * (posterior_alpha + posterior_beta + 1)
            )
            posterior_std = np.sqrt(posterior_var)

            # AUDIT POINT 2.1: Structural Veto (D6-Q2)
            # TeoriaCambio validation - caps Bayesian posterior â\211¤ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "structural_veto_threshold", 0.6) for impermissible links
            # Implements axiomatic-Bayesian fusion per Goertz & Mahoney 2012
            structural_violation = self._check_structural_violation(source, target)
            if structural_violation:
                # Deterministic veto: cap posterior at ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "structural_veto_threshold", 0.6) despite high semantic evidence
                original_posterior = posterior_mean
                posterior_mean = min(posterior_mean, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "structural_veto_threshold", 0.6))
```

```
                self.logger.warning(
                    f"STRUCTURAL VETO (D6-Q2): Link {source}â\206\222{target} violates causal hierarchy. "
                    f"Posterior capped from {original_posterior:.3f} to {posterior_mean:.3f}. "
                    f"Violation: {structural_violation}"
                )

            # Check convergence (require minimum evidence count)
            converged = (len(kl_divs) >= convergence_min_evidence and
                         len(kl_divs) > 0 and kl_divs[-1] < kl_threshold)
            final_kl = kl_divs[-1] if len(kl_divs) > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "final_kl_default", 0.0)

            # Add edge with posterior distribution
            self.graph.add_edge(
                source, target,
                logic=evidences[0]['logic'],
                keyword=evidences[0]['keyword'],
                strength=float(posterior_mean),
                posterior_mean=float(posterior_mean),
                posterior_std=float(posterior_std),
                posterior_alpha=float(posterior_alpha),
                posterior_beta=float(posterior_beta),
                kl_divergence=float(final_kl),
                converged=converged,
                evidence_count=len(evidences),
                structural_violation=structural_violation,
                veto_applied=structural_violation is not None
            )

            self.causal_chains.append({
                'source': source,
                'target': target,
                'logic': evidences[0]['logic'],
                'strength': float(posterior_mean),
                'evidence': [e['keyword'] for e in evidences],
                'posterior_mean': float(posterior_mean),
                'posterior_std': float(posterior_std),
                'kl_divergence': float(final_kl),
                'converged': converged
            })

        self.logger.info(f"Enlaces causales extraÃ-dos: {len(self.causal_chains)} "
                         f"(con inferencia Bayesiana)")

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance")
    def _calculate_semantic_distance(self, source: str, target: str) -> float:
        """
        Calculate semantic distance between nodes using spaCy embeddings

        PERFORMANCE NOTE: This method can be optimized with:
        1. Vectorized operations using numpy for batch processing
        2. Embedding caching to avoid recomputing spaCy vectors
        3. Async processing for large documents with many nodes
        4. Alternative: BERT/transformer embeddings for higher fidelity (SOTA)

        Current implementation prioritizes determinism over speed.
```

```
        Enable performance.cache_embeddings in config for production use.
        """
        try:
            source_node = self.nodes.get(source)
            target_node = self.nodes.get(target)

            if not source_node or not target_node:
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)

            # TODO: Implement embedding cache if performance.cache_embeddings is enabled
            # This would save ˜60% computation time on large documents

            # Use spaCy to get embeddings
            max_context = self.config.get_performance_setting('max_context_length') or 1000
            source_doc = self.nlp(source_node.text[:max_context])
            target_doc = self.nlp(target_node.text[:max_context])

            if source_doc.vector.any() and target_doc.vector.any():
                # Calculate cosine similarity (1 - distance)
                # PERFORMANCE NOTE: Could vectorize this with numpy.dot for batch operations
                similarity = 1 - cosine(source_doc.vector, target_doc.vector)
                return max(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "min_similarity", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "max_similarity", 1.0), similarity))

            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)
        except Exception:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior")
    def _calculate_type_transition_prior(self, source: str, target: str) -> float:
        """Calculate prior based on historical transition frequencies between goal types"""
        source_type = self.nodes[source].type
        target_type = self.nodes[target].type

        # Define transition probabilities based on logical flow
        # programa â\206\222 producto â\206\222 resultado â\206\222 impacto
        transition_priors = {
            ('programa', 'producto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'producto')", 0.85),
            ('producto', 'resultado'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('producto', 'resultado')", 0.80),
            ('resultado', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('resultado', 'impacto')", 0.75),
            ('programa', 'resultado'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'resultado')", 0.60),
            ('producto', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('producto', 'impacto')", 0.50),
            ('programa', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'impacto')", 0.30),
        }

        # Reverse transitions are less likely
        reverse_key = (target_type, source_type)
        if reverse_key in transition_priors:
            return transition_priors[reverse_key] * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "reverse_multiplier", 0.3)
```

```

        return transition_priors.get((source_type, target_type), ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "default", 0.40))

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._check_structural_violation")
    def _check_structural_violation(self, source: str, target: str) -> str | None:
        """
        AUDIT POINT 2.1: Structural Veto (D6-Q2)

        Check if causal link violates structural hierarchy based on TeoriaCambio axioms.
        Implements set-theoretic constraints per Goertz & Mahoney 2012.

        Returns:
            None if link is valid, otherwise a string describing the violation
        """
        source_type = self.nodes[source].type
        target_type = self.nodes[target].type

        # Define causal hierarchy levels (following TeoriaCambio axioms)
        # Lower levels cannot causally influence higher levels
        hierarchy_levels = {
            'programa': 1,
            'producto': 2,
            'resultado': 3,
            'impacto': 4
        }

        source_level = hierarchy_levels.get(source_type, 0)
        target_level = hierarchy_levels.get(target_type, 0)

        # Impermissible links: jumping more than 2 levels or reverse causation
        if target_level < source_level:
            # Reverse causation (e.g., Impacto â\206\222 Producto)
            return f"reverse_causation:{source_type}â\206\222{target_type}"

        if target_level - source_level > 2:
            # Skipping levels (e.g., Programa â\206\222 Impacto without intermediates)
            return f"level_skip:{source_type}â\206\222{target_type} (skips {target_level - source_level - 1} levels)"

        # Special case: Producto â\206\222 Impacto is impermissible (must go through Resultado)
        if source_type == 'producto' and target_type == 'impacto':
            return "missing_intermediate:productoâ\206\222impacto requires resultado"

        return None

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity")
    def _calculate_language_specificity(self, keyword: str, policy_area: str | None = None,
                                        context: str | None = None) -> float:
        """Assess specificity of causal language (epistemic certainty)

        Harmonic Front 3 - Enhancement 4: Language Specificity Assessment
        Enhanced to check policy-specific vocabulary (patrones_verificacion) for current
        Policy Area (P1â\200\223P10), not just generic causal keywords.

        For D6-Q5 (Contextual/Differential Focus): rewards use of specialized terminology
        that anchors intervention in social/cultural context (e.g., "catastro multipropÃ³sito",
```

```
        "reparaciÃ³n integral", "mujeres rurales", "guardia indÃgena").
        """
        # Strong causal indicators
        strong_indicators = ['causa', 'produce', 'genera', 'resulta en', 'conduce a']
        # Moderate indicators
        moderate_indicators = ['permite', 'contribuye', 'facilita', 'mediante', 'a travÃ©s de']
        # Weak indicators
        weak_indicators = ['con el fin de', 'para', 'porque']

        keyword_lower = keyword.lower()

        # Base score from causal indicators
        base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "base_score", 0.6) # Refactored
        if any(ind in keyword_lower for ind in strong_indicators):
            base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "strong_indicator_score", 0.9) # Refactored
        elif any(ind in keyword_lower for ind in moderate_indicators):
            base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "moderate_indicator_score", 0.7) # Refactored
        elif any(ind in keyword_lower for ind in weak_indicators):
            base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "weak_indicator_score", 0.5) # Refactored

        # HARMONIC FRONT 3 - Enhancement 4: Policy-specific vocabulary boost
        # Check for specialized terminology per policy area
        policy_area_vocabulary = {
            'P1': [  # Ordenamiento Territorial
                'catastro multipropÃ³sito', 'pot', 'pbot', 'eot', 'uaf', 'suelo de protecciÃ³n',
                'zonificaciÃ³n', 'uso del suelo', 'densificaciÃ³n', 'expansiÃ³n urbana'
            ],
            'P2': [  # VÃctimas y Paz
                'reparaciÃ³n integral', 'restituciÃ³n de tierras', 'vÃctimas del conflicto',
                'desplazamiento forzado', 'despojo', 'acciÃ³n integral', 'enfoque diferencial Ã©tnico',
                'construcciÃ³n de paz', 'reconciliaciÃ³n', 'memoria histÃ³rica'
            ],
            'P3': [  # Desarrollo Rural
                'mujeres rurales', 'extensiÃ³n agropecuaria', 'asistencia tÃ©cnica rural',
                'adecuaciÃ³n de tierras', 'comercializaciÃ³n campesina', 'economÃa campesina',
                'soberanÃa alimentaria', 'fondo de tierras'
            ],
            'P4': [  # Grupos Ã\211tnicos
                'guardia indÃgena', 'guardia cimarrona', 'territorios colectivos',
                'autoridades ancestrales', 'consulta previa', 'consentimiento libre',
                'medicina tradicional', 'sistema de salud propio indÃgena', 'jurisdicciÃ³n especial indÃgena'
            ],
            'P5': [  # Infraestructura y Conectividad
                'terciarias', 'vÃas terciarias', 'transporte intermodal', 'Ãºltimo kilÃ³metro',
                'conectividad digital', 'internet rural', 'electrificaciÃ³n rural'
            ],
            'P6': [  # Salud Rural
                'red hospitalaria', 'atenciÃ³n primaria', 'promotores de salud',
                'prevenciÃ³n de enfermedades tropicales', 'saneamiento bÃ¡sico', 'agua segura'
            ],
            'P7': [  # EducaciÃ³n Rural
                'escuela nueva', 'modelos flexibles', 'post-primaria rural',
```

```
                'educaciÃ³n propia', 'alfabetizaciÃ³n', 'deserciÃ³n escolar rural'
            ],
            'P8': [  # Vivienda y Habitabilidad
                'mejoramiento de vivienda rural', 'materiales locales', 'construcciÃ³n sostenible',
                'vivienda de interÃ©s social rural', 'titulaciÃ³n predial'
            ],
            'P9': [  # Medio Ambiente
                'pÃ¡ramos', 'humedales', 'Ã¡reas protegidas', 'corredores biolÃ³gicos',
                'servicios ecosistÃ©micos', 'pago por servicios ambientales', 'restauraciÃ³n ecolÃ³gica'
            ],
            'P10': [  # ReactivaciÃ³n EconÃ³mica
                'encadenamientos productivos', 'economÃa solidaria', 'cooperativas',
                'microcrÃ©dito', 'emprendimiento asociativo', 'fondo rotatorio'
            ]
        }

        # General contextual/differential focus vocabulary (D6-Q5)
        contextual_vocabulary = [
            'enfoque diferencial', 'enfoque de gÃ©nero', 'enfoque Ã©tnico',
            'acciÃ³n sin daÃ±o', 'pertinencia cultural', 'contexto territorial',
            'restricciones territoriales', 'barreras culturales', 'inequidad',
            'discriminaciÃ³n', 'exclusiÃ³n', 'vulnerabilidad', 'marginalidad',
            'ruralidad dispersa', 'aislamiento geogrÃ¡fico', 'baja densidad poblacional',
            'poblaciÃ³n dispersa', 'difÃcil acceso'
        ]

        # Check for policy-specific vocabulary boost
        specificity_boost = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "specificity_boost", 0.0) # Refactored
        text_to_check = (keyword_lower + ' ' + (context or '')).lower()

        if policy_area and policy_area in policy_area_vocabulary:
            for term in policy_area_vocabulary[policy_area]:
                if term.lower() in text_to_check:
                    specificity_boost = max(specificity_boost, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "policy_specificity_boost", 0.15))
                    self.logger.debug(f"Policy-specific term detected: '{term}' for {policy_area}")
                    break

        # Check for general contextual vocabulary (D6-Q5)
        for term in contextual_vocabulary:
            if term.lower() in text_to_check:
                specificity_boost = max(specificity_boost, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "contextual_specificity_boost", 0.10))
                self.logger.debug(f"Contextual term detected: '{term}'")
                break

        final_score = min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "final_score_max", 1.0), base_score + specificity_boost)

        return final_score

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence")
    def _assess_temporal_coherence(self, source: str, target: str) -> float:
        """Assess temporal coherence based on verb sequences"""
        source_node = self.nodes.get(source)
```

```
        target_node = self.nodes.get(target)

        if not source_node or not target_node:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "default", 0.5)

        # Extract verbs from entity-activity if available
        if source_node.entity_activity and target_node.entity_activity:
            source_verb = source_node.entity_activity.verb_lemma
            target_verb = target_node.entity_activity.verb_lemma

            # Define logical verb sequences
            verb_sequences = {
                ’diagnosticar’: 1, ’planificar’: 2, ’ejecutar’: 3, ’evaluar’: 4,
                ’diseÃ±ar’: 2, ’implementar’: 3, ’monitorear’: 4
            }

            source_seq = verb_sequences.get(source_verb, 5)
            target_seq = verb_sequences.get(target_verb, 5)

            if source_seq < target_seq:
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "in_sequence", 0.85)
            elif source_seq == target_seq:
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "same_sequence", 0.60)
            else:
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "reverse_sequence", 0.30)

        return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "default", 0.50)

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency")
    def _assess_financial_consistency(self, source: str, target: str) -> float:
        """Assess financial alignment between connected nodes"""
        source_node = self.nodes.get(source)
        target_node = self.nodes.get(target)

        if not source_node or not target_node:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "default", 0.5)

        source_budget = source_node.financial_allocation
        target_budget = target_node.financial_allocation

        if source_budget and target_budget:
            # Check if budgets are aligned (target should be <= source)
            ratio = target_budget / source_budget if source_budget > 0 else 0

            if ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_min", 0.1) <= ratio <= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_max", 1.0):
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_score", 0.85)
            elif ratio > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_max", 1.0) and ratio <= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "slightly_inconsistent_max", 1.5):
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "slightly_inconsistent_score", 0.60)
            else:
                return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "very_inconsistent_score", 0.30)

        return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "default", 0.50)
```

```

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_textual_proximity")
    def _calculate_textual_proximity(self, source: str, target: str, text: str) -> float:
        """Calculate how often node IDs appear together in text windows"""
        window_size = 200  # characters
        co_occurrences = 0
        total_windows = 0

        source_positions = [m.start() for m in re.finditer(re.escape(source), text, re.IGNORECASE)]
        target_positions = [m.start() for m in re.finditer(re.escape(target), text, re.IGNORECASE)]

        for source_pos in source_positions:
            total_windows += 1
            for target_pos in target_positions:
                if abs(source_pos - target_pos) <= window_size:
                    co_occurrences += 1
                    break

        if total_windows > 0:
            proximity_score = co_occurrences / total_windows
            return proximity_score

        return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_textual_proximity", "default", 0.5)

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._initialize_prior")
    def _initialize_prior(self, source: str, target: str) -> tuple[float, float, float]:
        """Initialize prior distribution for causal link"""
        # Use type transition as base prior
        type_prior = self._calculate_type_transition_prior(source, target)

        # Beta distribution parameters - now externalized
        prior_alpha = self.config.get_bayesian_threshold('prior_alpha')
        prior_beta = self.config.get_bayesian_threshold('prior_beta')

        # Adjust based on type transition
        prior_mean = type_prior
        prior_strength = prior_alpha + prior_beta

        adjusted_alpha = prior_mean * prior_strength
        adjusted_beta = (1 - prior_mean) * prior_strength

        return prior_mean, adjusted_alpha, adjusted_beta

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood")
    def _calculate_composite_likelihood(self, evidence: dict[str, Any]) -> float:
        """Calculate composite likelihood from multiple evidence components

        Enhanced with:
        - Nonlinear transformation rewarding triangulation
        - Evidence diversity verification across analytical domains
        """
        # Weight different evidence types
        weights = {
            'semantic_distance': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "semantic_distance_weight", 0.25),
            'type_transition_prior': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "type_transit
```

```
ion_prior_weight", 0.20),
            'language_specificity': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "language_specificity_weight", 0.20),
            'temporal_coherence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "temporal_coherence_weight", 0.15),
            'financial_consistency': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "financial_consistency_weight", 0.10),
            'textual_proximity': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "textual_proximity_weight", 0.10)
        }

        # Basic weighted average
        likelihood = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "likelihood", 0.0) # Refactored
        evidence_count = 0
        domain_diversity = set()

        for component, weight in weights.items():
            if component in evidence:
                likelihood += evidence[component] * weight
                evidence_count += 1

                # Track evidence diversity across domains
                if component in ['semantic_distance', 'textual_proximity']:
                    domain_diversity.add('semantic')
                elif component in ['temporal_coherence']:
                    domain_diversity.add('temporal')
                elif component in ['financial_consistency']:
                    domain_diversity.add('financial')
                elif component in ['type_transition_prior', 'language_specificity']:
                    domain_diversity.add('structural')

        # Triangulation bonus: Exponentially reward multiple independent observations
        # D6-Q4/Q5 (Adaptiveness/Context) - evidence across different analytical domains
        diversity_count = len(domain_diversity)
        if diversity_count >= 3:
            # Strong triangulation across semantic, temporal, and financial domains
            triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_3", 1.15)
        elif diversity_count == 2:
            # Moderate triangulation
            triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_2", 1.05)
        else:
            # Weak or no triangulation
            triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_default", 1.0) # Refactored

        # Apply nonlinear transformation
        enhanced_likelihood = min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "enhanced_likelihood_max", 1.0), likelihood * triangulation_bonus)

        # Penalty for insufficient evidence diversity
        if evidence_count < 3:
            enhanced_likelihood *= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "insufficient_evidence_penalty", 0.85)
```

```
        return enhanced_likelihood

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy")
    def _build_type_hierarchy(self) -> None:
        """Build hierarchy based on goal types"""

        nodes_by_type: dict[str, list[str]] = defaultdict(list)
        for node_id in self.graph.nodes():
            node_type = self.graph.nodes[node_id].get('type', 'programa')
            nodes_by_type[node_type].append(node_id)

        # Connect productos to programas
        for prod in nodes_by_type.get('producto', []):
            for prog in nodes_by_type.get('programa', []):
                if not self.graph.has_edge(prog, prod):
                    self.graph.add_edge(prog, prod, logic='inferido', strength=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy", "inferred_strength", 0.5))

        # Connect resultados to productos
        for res in nodes_by_type.get('resultado', []):
            for prod in nodes_by_type.get('producto', []):
                if not self.graph.has_edge(prod, res):
                    self.graph.add_edge(prod, res, logic='inferido', strength=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy", "inferred_strength", 0.5))

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence")
    def _calculate_confidence(self, node: MetaNode, link_text: str = "") -> float:
        """
        Calculate confidence score for a causal link.

        Args:
            node: The node to calculate confidence for
            link_text: Optional text describing the causal link

        Returns:
            Confidence score between 0 and 1
        """
        confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "confidence", 0.5) # Refactored

        # Increase confidence if node has quantitative targets
        if node.target and node.baseline:
            try:
                float(str(node.target).replace(',', '').replace('%', ''))
                confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "quantitative_target_bonus", 0.2)
            except (ValueError, TypeError):
                pass

        # Increase confidence if text has causal indicators
        if link_text:
            causal_words = ['porque', 'debido', 'mediante', 'a travÃ©s', 'permite', 'genera', 'produce']
            if any(word in link_text.lower() for word in causal_words):
                confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "causal_indicator_bonus", 0.15)

```

```
        # Increase confidence based on rigor status
        if hasattr(node, ’rigor_status’):
            if node.rigor_status == ’fuerte’:
                confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "strong_rigor_bonus", 0.15)
            elif node.rigor_status == ’dÃ©bil’:
                confidence -= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "weak_rigor_penalty", 0.1)

        return min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "max_confidence", 1.0), max(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "min_confidence", 0.0), confidence))

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._classify_goal_type")
    def _classify_goal_type(self, text: str) -> str:
        """
        Classify the type of a goal based on its text.

        Args:
            text: Goal text to classify

        Returns:
            Goal type (programa, producto, resultado, impacto)
        """
        text_lower = text.lower()

        # Keywords for each type
        if any(word in text_lower for word in [’programa’, ’lÃ-nea estratÃ©gica’, ’componente’, ’eje’]):
            return ’programa’
        elif any(word in text_lower for word in [’producto’, ’servicio’, ’bien’, ’actividad’]):
            return ’producto’
        elif any(word in text_lower for word in [’resultado’, ’efecto’, ’cambio’, ’mejora’]):
            return ’resultado’
        elif any(word in text_lower for word in [’impacto’, ’transformaciÃ³n’, ’desarrollo’, ’bienestar’]):
            return ’impacto’
        # Default classification based on position and complexity
        elif len(text) < 100:
            return ’producto’
        else:
            return ’resultado’

    @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_justifications")
    def _extract_causal_justifications(self, text: str) -> list[dict[str, Any]]:
        """
        Extract causal justifications from text.

        Args:
            text: Text to extract justifications from

        Returns:
            List of justifications with text and confidence
        """
        justifications = []

        # Patterns that indicate causal justifications
        patterns = [
            r’porque\s+([^.]+)’,
            r’debido\s+a\s+([^.]+)’,
            r’mediante\s+([^.]+)’,
```

```
            r'a\s+travÃ©s\s+de\s+([^.]+)',
            r'se\s+logra\s+mediante\s+([^.]+)',
            r'permite\s+([^.]+)',
            r'genera\s+([^.]+)',
        ]

        for pattern in patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                justification_text = match.group(1).strip()
                justifications.append({
                    'text': justification_text,
                    'confidence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_justifications", "confidence", 0.7),
                    'type': 'causal_explanation'
                })

        return justifications

class MechanismPartExtractor:
    """Extract Entity-Activity pairs for mechanism parts"""

    def __init__(self, config: ConfigLoader, nlp_model: spacy.Language) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.nlp = nlp_model
        self.entity_aliases = config.get('entity_aliases', {})

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor.extract_entity_activity")
    def extract_entity_activity(self, text: str) -> EntityActivity | None:
        """Extract Entity-Activity tuple from text"""
        doc = self.nlp(text)

        # Find main verb (activity)
        main_verb = None
        for token in doc:
            if token.pos_ == 'VERB' and token.dep_ in ['ROOT', 'ccomp']:
                main_verb = token
                break

        if not main_verb:
            return None

        # Find subject entity
        entity = None
        for child in main_verb.children:
            if child.dep_ in ['nsubj', 'nsubjpass']:
                entity = self._normalize_entity(child.text)
                break

        if not entity:
            # Try to find entity from NER
            for ent in doc.ents:
                if ent.label_ in ['ORG', 'PER']:
                    entity = self._normalize_entity(ent.text)
                    break
```

```

        if entity and main_verb:
            return EntityActivity(
                entity=entity,
                activity=main_verb.text,
                verb_lemma=main_verb.lemma_,
                confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor.extract_entity_activity", "confidence", 0.85) # Refactored
            )

        return None

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._normalize_entity")
    def _normalize_entity(self, entity: str) -> str:
        """Normalize entity name using aliases"""
        entity_upper = entity.upper().strip()
        return self.entity_aliases.get(entity_upper, entity)

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence")
    def _calculate_ea_confidence(self, entity: str, activity: str, context: str = "") -> float:
        """
        Calculate confidence for an entity-activity pair.

        Args:
            entity: Entity text
            activity: Activity text
            context: Surrounding context

        Returns:
            Confidence score between 0 and 1
        """
        confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "confidence", 0.5) # Refactored

        # Higher confidence if entity is in known aliases
        if entity.upper() in self.entity_aliases:
            confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "known_entity_bonus", 0.2)

        # Higher confidence if activity is a strong verb
        strong_verbs = ['ejecutar', 'implementar', 'desarrollar', 'gestionar', 'coordinar']
        if any(verb in activity.lower() for verb in strong_verbs):
            confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "strong_verb_bonus", 0.15)

        # Higher confidence if there's clear grammatical connection in context
        if entity in context and activity in context:
            confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "grammatical_connection_bonus", 0.15)

        return min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "max_confidence", 1.0), confidence)

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._find_action_verb")
    def _find_action_verb(self, text: str) -> str | None:
        """
```

```
        Find the main action verb in text.

        Args:
            text: Text to analyze

        Returns:
            Main action verb or None
        """
        doc = self.nlp(text)

        # Find main verb
        for token in doc:
            if token.pos_ == 'VERB' and token.dep_ in ['ROOT', 'ccomp', 'xcomp']:
                return token.text

        # Fallback: any verb
        for token in doc:
            if token.pos_ == 'VERB':
                return token.text

        return None

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._find_subject_entity")
    def _find_subject_entity(self, text: str) -> str | None:
        """
        Find the subject entity in text.

        Args:
            text: Text to analyze

        Returns:
            Subject entity or None
        """
        doc = self.nlp(text)

        # Find subject
        for token in doc:
            if token.dep_ in ['nsubj', 'nsubjpass']:
                return self._normalize_entity(token.text)

        # Try NER
        for ent in doc.ents:
            if ent.label_ in ['ORG', 'PER', 'GPE']:
                return self._normalize_entity(ent.text)

        return None

    @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._validate_entity_activity")
    def _validate_entity_activity(self, entity: str, activity: str) -> bool:
        """
        Validate that an entity-activity pair makes sense.

        Args:
            entity: Entity text
            activity: Activity text

```

```
        Returns:
            True if valid pair
        """
        # Basic validation
        if not entity or not activity:
            return False

        # Entity should not be too short or generic
        if len(entity) < 3 or entity.lower() in ['el', 'la', 'los', 'las', 'un', 'una']:
            return False

        # Activity should be a reasonable verb
        return not len(activity) < 3

class FinancialAuditor:
    """Financial traceability and auditing"""

    def __init__(self, config: ConfigLoader) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.financial_data: dict[str, dict[str, float]] = {}
        self.unit_costs: dict[str, float] = {}
        self.successful_parses = 0
        self.failed_parses = 0
        self.d3_q3_analysis: dict[str, Any] = {}  # Harmonic Front 3 - D3-Q3 metrics

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor.trace_financial_allocation")
    def trace_financial_allocation(self, tables: list[pd.DataFrame],
                                   nodes: dict[str, MetaNode],
                                   graph: nx.DiGraph | None = None) -> dict[str, float]:
        """Trace financial allocations to programs/goals

        Harmonic Front 3 - Enhancement 5: Single-Case Counterfactual Budget Check
        Incorporates logic from single-case counterfactuals to test minimal sufficiency.
        For D3-Q3 (Traceability/Resources): checks if resource X (BPIN code) were removed,
        would the mechanism (Product) still execute? Only boosts budget traceability score
        if allocation is tied to a specific project.
        """
        for i, table in enumerate(tables):
            try:
                self.logger.info(f"Procesando tabla financiera {i + 1}/{len(tables)}")
                self._process_financial_table(table, nodes)
                self.successful_parses += 1
            except Exception as e:
                self.logger.error(f"Error procesando tabla financiera {i + 1}: {e}")
                self.failed_parses += 1
                continue

        # HARMONIC FRONT 3 - Enhancement 5: Counterfactual sufficiency check
        if graph is not None:
            self._perform_counterfactual_budget_check(nodes, graph)

        self.logger.info(f"Asignaciones financieras trazadas: {len(self.financial_data)}")
        self.logger.info(f"Tablas parseadas exitosamente: {self.successful_parses}, "
                         f"Fallidas: {self.failed_parses}")
        return self.unit_costs
```

```

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._process_financial_table")
    def _process_financial_table(self, table: pd.DataFrame,
                                 nodes: dict[str, MetaNode]) -> None:
        """Process a single financial table"""
        # Try to identify relevant columns
        amount_pattern = re.compile(
            self.config.get('patterns.financial_headers', r'PRESUPUESTO|VALOR|MONTO'),
            re.IGNORECASE
        )
        program_pattern = re.compile(r'PROGRAMA|META|CÃ\223DIGO', re.IGNORECASE)

        amount_col = None
        program_col = None

        # Search in column names
        for col in table.columns:
            col_str = str(col)
            if amount_pattern.search(col_str) and not amount_col:
                amount_col = col
            if program_pattern.search(col_str) and not program_col:
                program_col = col

        # If not found in column names, search in first row
        if not amount_col or not program_col:
            first_row = table.iloc[0]
            for i, val in enumerate(first_row):
                val_str = str(val)
                if amount_pattern.search(val_str) and not amount_col:
                    amount_col = i
                    table.columns = table.iloc[0]
                    table = table[1:]
                if program_pattern.search(val_str) and not program_col:
                    program_col = i
                    table.columns = table.iloc[0]
                    table = table[1:]

        if amount_col is None or program_col is None:
            self.logger.warning("No se encontraron columnas financieras relevantes")
            return

        for _, row in table.iterrows():
            try:
                program_id = str(row[program_col]).strip().upper()
                amount = self._parse_amount(row[amount_col])

                if amount and program_id:
                    matched_node = self._match_program_to_node(program_id, nodes)
                    if matched_node:
                        self.financial_data[matched_node] = {
                            'allocation': amount,
                            'source': 'budget_table'
                        }

                        # Update node
                        nodes[matched_node].financial_allocation = amount
```

```

                        # Calculate unit cost if possible
                        node = nodes.get(matched_node)
                        if node and node.target:
                            try:
                                target_val = float(str(node.target).replace(',', '').replace('%', ''))
                                if target_val > 0:
                                    unit_cost = amount / target_val
                                    self.unit_costs[matched_node] = unit_cost
                                    nodes[matched_node].unit_cost = unit_cost
                            except (ValueError, TypeError):
                                pass

            except Exception as e:
                self.logger.debug(f"Error procesando fila financiera: {e}")
                continue

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._parse_amount")
    def _parse_amount(self, value: Any) -> float | None:
        """Parse monetary amount from various formats"""
        if pd.isna(value):
            return None

        try:
            clean_value = str(value).replace('$', '').replace(',', '').replace(' ', '').replace('.', '')
            # Handle millions/thousands notation
            if 'M' in clean_value.upper() or 'MILLONES' in clean_value.upper():
                clean_value = clean_value.upper().replace('M', '').replace('ILLONES', '')
                return float(clean_value) * 1_000_000
            return float(clean_value)
        except (ValueError, TypeError):
            return None

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node")
    def _match_program_to_node(self, program_id: str,
                               nodes: dict[str, MetaNode]) -> str | None:
        """Match program ID to existing node using fuzzy matching

        Enhanced for D1-Q3 / D3-Q3 Financial Traceability:
        - Implements confidence penalty if fuzzy match ratio < 100
        - Reduces node.financial_allocation confidence by 15% for imperfect matches
        - Tracks match quality for overall financial traceability scoring
        """
        if program_id in nodes:
            # Perfect match - no penalty
            return program_id

        # Try fuzzy matching
        best_match = process.extractOne(
            program_id,
            nodes.keys(),
            scorer=fuzz.ratio,
            score_cutoff=80
        )

        if best_match:
```

```
            matched_node_id = best_match[0]
            match_ratio = best_match[1]

            # D1-Q3 / D3-Q3: Apply confidence penalty for non-perfect matches
            if match_ratio < 100:
                penalty_factor = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node", "penalty_factor", 0.85) # Refactored
                node = nodes[matched_node_id]

                # Track original allocation before penalty
                if not hasattr(node, '_original_financial_allocation'):
                    node._original_financial_allocation = node.financial_allocation

                # Apply penalty to financial allocation confidence
                if node.financial_allocation:
                    penalized_allocation = node.financial_allocation * penalty_factor
                    self.logger.debug(
                        f"Fuzzy match penalty applied to {matched_node_id}: "
                        f"ratio={match_ratio}, penalty={penalty_factor:.2f}, "
                        f"allocation {node.financial_allocation:.0f} -> {penalized_allocation:.0f}"
                    )
                    node.financial_allocation = penalized_allocation

                # Store match confidence for D1-Q3 / D3-Q3 scoring
                if not hasattr(node, 'financial_match_confidence'):
                    node.financial_match_confidence = match_ratio / ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node", "match_confidence_divisor", 100.0)
                else:
                    # Average if multiple matches
                    node.financial_match_confidence = (node.financial_match_confidence + match_ratio / ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node", "match_confidence_divisor", 100.0)) / 2

            return matched_node_id

        return None

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check")
    def _perform_counterfactual_budget_check(self, nodes: dict[str, MetaNode],
                                             graph: nx.DiGraph) -> None:
        """
        Harmonic Front 3 - Enhancement 5: Counterfactual Sufficiency Test for D3-Q3

        Tests minimal sufficiency: if resource X (BPIN code) were removed, would the
        mechanism (Product) still execute? Only boosts budget traceability score if
        allocation is tied to a specific project.

        For D3-Q3 (Traceability/Resources): ensures funding is necessary for the mechanism
        and prevents false positives from generic or disconnected budget entries.
        """
        d3_q3_scores = {}

        for node_id, node in nodes.items():
            if node.type != 'producto':
                continue

            # Check if node has financial allocation
```

```
            has_budget = node.financial_allocation is not None and node.financial_allocation > 0

            # Check if node has entity-activity (mechanism)
            has_mechanism = node.entity_activity is not None

            # Check if node has dependencies (successors in graph)
            successors = list(graph.successors(node_id)) if graph.has_node(node_id) else []
            has_dependencies = len(successors) > 0

            # Counterfactual test: Would mechanism still execute without this budget?
            # Check if there are alternative funding sources or generic allocations
            financial_source = self.financial_data.get(node_id, {}).get('source', 'unknown')
            is_specific_allocation = financial_source == 'budget_table'  # From specific table entry

            # Calculate counterfactual necessity score
            # High score = budget is necessary for execution
            # Low score = budget may be generic/disconnected
            necessity_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "necessity_score", 0.0) # Refactored

            if has_budget and has_mechanism:
                necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "budget_mechanism_bonus", 0.40)  # Budget + mechanism present

            if has_budget and has_dependencies:
                necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "budget_dependency_bonus", 0.30)  # Budget supports downstream goals

            if is_specific_allocation:
                necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "specific_allocation_bonus", 0.30)  # Specific allocation (not generic)

            # D3-Q3 quality criteria
            d3_q3_quality = 'insuficiente'
            if necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "excellent_threshold", 0.85):
                d3_q3_quality = 'excelente'
            elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "good_threshold", 0.70):
                d3_q3_quality = 'bueno'
            elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "acceptable_threshold", 0.50):
                d3_q3_quality = 'aceptable'

            d3_q3_scores[node_id] = {
                'necessity_score': necessity_score,
                'd3_q3_quality': d3_q3_quality,
                'has_budget': has_budget,
                'has_mechanism': has_mechanism,
                'has_dependencies': has_dependencies,
                'is_specific_allocation': is_specific_allocation,
                'counterfactual_sufficient': necessity_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "sufficient_threshold", 0.50),  # Would still execute without budget
                'budget_necessary': necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "necessary_threshold", 0.70)  # Budget is necessary
            }
```

```

            # Store in node for later retrieval
            node.audit_flags = node.audit_flags or []
            if necessity_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "sufficient_threshold", 0.50):
                node.audit_flags.append('budget_not_necessary')
                self.logger.warning(
                    f"D3-Q3: {node_id} may execute without allocated budget (score={necessity_score:.2f})")
            elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "excellent_threshold", 0.85):
                node.audit_flags.append('budget_well_traced')
                self.logger.info(f"D3-Q3: {node_id} has well-traced, necessary budget (score={necessity_score:.2f})")

        # Store aggregate D3-Q3 metrics
        self.d3_q3_analysis = {
            'node_scores': d3_q3_scores,
            'total_products_analyzed': len(d3_q3_scores),
            'well_traced_count': sum(1 for s in d3_q3_scores.values() if s['d3_q3_quality'] == 'excelente'),
            'average_necessity_score': sum(s['necessity_score'] for s in d3_q3_scores.values()) / max(len(d3_q3_scores),
                                                                                                      1)
        }

        self.logger.info(f"D3-Q3 Counterfactual Budget Check completed: "
                         f"{self.d3_q3_analysis['well_traced_count']}/{len(d3_q3_scores)} "
                         f"products with excellent traceability")

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency")
    def _calculate_sufficiency(self, allocation: float, target: float) -> float:
        """
        Calculate if financial allocation is sufficient for target.

        Args:
            allocation: Financial allocation amount
            target: Target value

        Returns:
            Sufficiency ratio (ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "auto_param_L2208_31", 1.0) = exactly sufficient, >ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "auto_param_L2208_58", 1.0) = oversufficient)
        """
        if not target or target == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "default", 0.0)

        # Calculate unit cost implied by allocation and target
        allocation / target

        # Compare with historical/expected unit costs if available
        # For now, return simple ratio
        return allocation / target if target > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "default", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._detect_allocation_gaps")
    def _detect_allocation_gaps(self, nodes: dict[str, MetaNode]) -> list[dict[str, Any]]:
        """
        Detect gaps in financial allocations.

```

```
        Args:
            nodes: Dictionary of nodes

        Returns:
            List of detected gaps
        """
        gaps = []

        for node_id, node in nodes.items():
            # Check for missing allocation
            if node.type in [’producto’, ’programa’] and not node.financial_allocation:
                gaps.append({
                    ’node_id’: node_id,
                    ’type’: ’missing_allocation’,
                    ’severity’: ’high’,
                    ’message’: f"No financial allocation for {node.type} {node_id}"
                })

            # Check for insufficient allocation
            if node.financial_allocation and node.target:
                try:
                    target_val = float(str(node.target).replace(’,’, ’’).replace(’%’, ’’))
                    if target_val > 0:
                        sufficiency = self._calculate_sufficiency(node.financial_allocation, target_val)
                        if sufficiency < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._detect_allocation_gaps", "sufficiency_threshold", 0.5):
                            gaps.append({
                                ’node_id’: node_id,
                                ’type’: ’insufficient_allocation’,
                                ’severity’: ’medium’,
                                ’message’: f"Low sufficiency ratio {sufficiency:.2f} for {node_id}",
                                ’sufficiency’: sufficiency
                            })
                except (ValueError, TypeError):
                    pass

        return gaps

    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._match_goal_to_budget")
    def _match_goal_to_budget(self, goal_text: str, budget_entries: list[dict[str, Any]]) -> dict[str, Any] | None:
        """
        Match a goal to budget entries.

        Args:
            goal_text: Goal text to match
            budget_entries: List of budget entries

        Returns:
            Best matching budget entry or None
        """
        if not budget_entries:
            return None

        # Extract potential identifiers from goal text
        goal_words = set(goal_text.lower().split())

```

```
        best_match = None
        best_score = 0

        for entry in budget_entries:
            entry_text = str(entry.get('description', '')).lower()
            entry_words = set(entry_text.split())

            # Calculate overlap
            overlap = len(goal_words & entry_words)
            score = overlap / max(len(goal_words), len(entry_words), 1)

            if score > best_score and score > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_goal_to_budget", "score_threshold", 0.3):  # Minimum threshold
                best_score = score
                best_match = entry

        return best_match

class OperationalizationAuditor:
    """Audit operationalization quality"""

    def __init__(self, config: ConfigLoader) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.verb_sequences = config.get('verb_sequences', {})
        self.audit_results: dict[str, AuditResult] = {}
        self.sequence_warnings: list[str] = []

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor.audit_evidence_traceability")
    def audit_evidence_traceability(self, nodes: dict[str, MetaNode]) -> dict[str, AuditResult]:
        """Audit evidence traceability for all nodes

        Enhanced with D3-Q1 Ficha TÃ©cnica validation:
        - Cross-checks baseline/target against extracted quantitative_claims
        - Verifies DNP INDICATOR_STRUCTURE compliance for producto nodes
        - Scores 'Excelente' only if â\211¥80% of productos pass full audit
        """
        # Import for quantitative claims extraction
        try:
            from contradiction_deteccion import PolicyContradictionDetectorV2
            has_detector = True
        except ImportError:
            has_detector = False
            self.logger.warning("PolicyContradictionDetectorV2 not available for quantitative claims validation")

        producto_nodes_count = 0
        producto_nodes_passed = 0

        for node_id, node in nodes.items():
            result: AuditResult = {
                'passed': True,
                'warnings': [],
                'errors': [],
                'recommendations': []
            }

```

```
            # Track producto nodes for D3-Q1 scoring
            if node.type == 'producto':
                producto_nodes_count += 1

            # Extract quantitative claims from node text if detector available
            quantitative_claims = []
            if has_detector:
                try:
                    # Create temporary detector instance
                    detector = PolicyContradictionDetectorV2(device='cpu')
                    quantitative_claims = detector._extract_structured_quantitative_claims(node.text)
                except Exception as e:
                    self.logger.debug(f"Could not extract quantitative claims: {e}")

            # Check baseline
            baseline_valid = False
            if not node.baseline or str(node.baseline).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
                result['errors'].append(f"LÃ-nea base no definida para {node_id}")
                result['passed'] = False
                node.rigor_status = 'dÃ©bil'
                node.audit_flags.append('sin_linea_base')
            else:
                baseline_valid = True
                # Cross-check baseline against quantitative claims (D3-Q1)
                if quantitative_claims:
                    baseline_in_claims = any(
                        claim.get('type') in ['indicator', 'target', 'percentage', 'beneficiaries']
                        for claim in quantitative_claims
                    )
                    if not baseline_in_claims:
                        result['warnings'].append(f"LÃ-nea base no verificada en claims cuantitativos para {node_id}")

            # Check target
            target_valid = False
            if not node.target or str(node.target).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
                result['errors'].append(f"Meta no definida para {node_id}")
                result['passed'] = False
                node.rigor_status = 'dÃ©bil'
                node.audit_flags.append('sin_meta')
            else:
                target_valid = True
                # Cross-check target against quantitative claims (D3-Q1)
                if quantitative_claims:
                    meta_in_claims = any(
                        claim.get('type') == 'target' or 'meta' in claim.get('context', '').lower()
                        for claim in quantitative_claims
                    )
                    if not meta_in_claims:
                        result['warnings'].append(f"Meta no verificada en claims cuantitativos para {node_id}")

            # D3-Q1 Ficha TÃ©cnica compliance check for producto nodes
            if node.type == 'producto':
                # Check if has all minimum DNP INDICATOR_STRUCTURE elements
                has_complete_ficha = (
                        baseline_valid and
                        target_valid and
```

```
                        ’sin_linea_base’ not in node.audit_flags and
                        ’sin_meta’ not in node.audit_flags
                )

                if has_complete_ficha and quantitative_claims:
                    # Node passes D3-Q1 compliance
                    producto_nodes_passed += 1
                    result[’recommendations’].append(f"D3-Q1 Ficha TÃ©cnica completa para {node_id}")
                elif has_complete_ficha:
                    # Has baseline/target but no quantitative claims verification
                    producto_nodes_passed += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor.audit_evidence_traceability", "partial_credit", 0.5)  # Partial credit
                    result[’warnings’].append(f"D3-Q1 parcial: Ficha bÃ¡sica sin verificaciÃ³n cuantitativa en {node_id}")

            # Check responsible entity
            if not node.responsible_entity:
                result[’warnings’].append(f"Entidad responsable no identificada para {node_id}")
                node.audit_flags.append(’sin_responsable’)

            # Check financial traceability
            if not node.financial_allocation:
                result[’warnings’].append(f"Sin trazabilidad financiera para {node_id}")
                node.audit_flags.append(’sin_presupuesto’)

            # Set rigor status if passed all checks
            if result[’passed’] and len(result[’warnings’]) == 0:
                node.rigor_status = ’fuerte’

            self.audit_results[node_id] = result

        # Calculate D3-Q1 compliance score
        if producto_nodes_count > 0:
            d3_q1_compliance_pct = (producto_nodes_passed / producto_nodes_count) * 100
            self.logger.info(f"D3-Q1 Ficha TÃ©cnica Compliance: {d3_q1_compliance_pct:.1f}% "
                             f"({producto_nodes_passed}/{producto_nodes_count} productos)")

            if d3_q1_compliance_pct >= 80:
                self.logger.info("D3-Q1 Score: EXCELENTE (â\211¥80% productos con Ficha TÃ©cnica completa)")
            elif d3_q1_compliance_pct >= 60:
                self.logger.info("D3-Q1 Score: BUENO (60-80% compliance)")
            else:
                self.logger.warning("D3-Q1 Score: INSUFICIENTE (<60% compliance)")

        passed_count = sum(1 for r in self.audit_results.values() if r[’passed’])
        self.logger.info(f"AuditorÃ-a de trazabilidad: {passed_count}/{len(nodes)} nodos aprobados")

        return self.audit_results

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor.audit_sequence_logic")
    def audit_sequence_logic(self, graph: nx.DiGraph) -> list[str]:
        """Audit logical sequence of activities"""
        warnings = []

        # Group nodes by program
        programs: dict[str, list[str]] = defaultdict(list)
        for node_id in graph.nodes():
```

```
            node_data = graph.nodes[node_id]
            if node_data.get('type') == 'programa':
                for successor in graph.successors(node_id):
                    if graph.nodes[successor].get('type') == 'producto':
                        programs[node_id].append(successor)

        # Check sequence within each program
        for program_id, product_goals in programs.items():
            if len(product_goals) < 2:
                continue

            activities = []
            for goal_id in product_goals:
                node = graph.nodes[goal_id]
                ea = node.get('entity_activity')
                if ea and isinstance(ea, dict):
                    verb = ea.get('verb_lemma', '')
                    sequence_num = self.verb_sequences.get(verb, 999)
                    activities.append((goal_id, verb, sequence_num))

            # Check for sequence violations
            activities.sort(key=lambda x: x[2])
            for i in range(len(activities) - 1):
                if activities[i][2] > activities[i + 1][2]:
                    warning = (f"ViolaciÃ³n de secuencia en {program_id}: "
                               f"{activities[i][1]} ({activities[i][0]}) "
                               f"antes de {activities[i + 1][1]} ({activities[i + 1][0]})")
                    warnings.append(warning)
                    self.logger.warning(warning)

        self.sequence_warnings = warnings
        return warnings

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit")
    def bayesian_counterfactual_audit(self, nodes: dict[str, MetaNode],
                                      graph: nx.DiGraph,
                                      historical_data: dict[str, Any] | None = None,
                                      pdet_alignment: float | None = None) -> dict[str, Any]:
        """
        AGUJA III: El Auditor Contrafactual Bayesiano
        Perform counterfactual audit using Bayesian causal reasoning

        Harmonic Front 3: Enhanced to consume pdet_alignment scores for D4-Q5 and D5-Q4 integration
        """
        self.logger.info("Iniciando auditorÃa contrafactual Bayesiana...")

        # Build implicit Structural Causal Model (SCM)
        scm_dag = self._build_normative_dag()

        # Initialize historical priors
        if historical_data is None:
            historical_data = self._get_default_historical_priors()

        # Audit results by layers
        layer1_results = self._audit_direct_evidence(nodes, scm_dag, historical_data)
        layer2_results = self._audit_causal_implications(nodes, graph, layer1_results)
```

```
        layer3_results = self._audit_systemic_risk(nodes, graph, layer1_results, layer2_results, pdet_alignment)

        # Generate optimal remediation recommendations
        recommendations = self._generate_optimal_remediations(
            layer1_results, layer2_results, layer3_results
        )

        audit_report = {
            'direct_evidence': layer1_results,
            'causal_implications': layer2_results,
            'systemic_risk': layer3_results,
            'recommendations': recommendations,
            'summary': {
                'total_nodes': len(nodes),
                'critical_omissions': sum(1 for r in layer1_results.values()
                                          if r.get('omission_severity') == 'critical'),
                'expected_success_probability': layer3_results.get('success_probability', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit", "success_probability_default", 0.0)),
                'risk_score': layer3_results.get('risk_score', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit", "risk_score_default", 0.0))
            }
        }

        self.logger.info(f"AuditorÃa contrafactual completada: "
                         f"{audit_report['summary']['critical_omissions']} omisiones crÃticas detectadas")

        return audit_report

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._build_normative_dag")
    def _build_normative_dag(self) -> nx.DiGraph:
        """Build normative DAG of expected relationships in well-formed plans"""
        dag = nx.DiGraph()

        # Define normative structure
        # Each goal type should have these attributes
        dag.add_node('baseline', type='required_attribute')
        dag.add_node('target', type='required_attribute')
        dag.add_node('entity', type='required_attribute')
        dag.add_node('budget', type='recommended_attribute')
        dag.add_node('mechanism', type='recommended_attribute')
        dag.add_node('timeline', type='optional_attribute')
        dag.add_node('risk_factors', type='optional_attribute')

        # Causal relationships
        dag.add_edge('baseline', 'target', relation='defines_gap')
        dag.add_edge('entity', 'mechanism', relation='executes')
        dag.add_edge('budget', 'mechanism', relation='enables')
        dag.add_edge('mechanism', 'target', relation='achieves')
        dag.add_edge('risk_factors', 'target', relation='threatens')

        return dag

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors")
    def _get_default_historical_priors(self) -> dict[str, Any]:
        """Get default historical priors if no data is available"""
        return {
```

```
            'entity_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "entity_presence_success_rate", 0.94),
            'baseline_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "baseline_presence_success_rate", 0.89),
            'target_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "target_presence_success_rate", 0.92),
            'budget_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "budget_presence_success_rate", 0.78),
            'mechanism_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "mechanism_presence_success_rate", 0.65),
            'complete_documentation_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "complete_documentation_success_rate", 0.82),
            'node_type_success_rates': {
                'producto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "producto_success_rate", 0.85),
                'resultado': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "resultado_success_rate", 0.72),
                'impacto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "impacto_success_rate", 0.58)
            }
        }

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence")
    def _audit_direct_evidence(self, nodes: dict[str, MetaNode],
                               scm_dag: nx.DiGraph,
                               historical_data: dict[str, Any]) -> dict[str, dict[str, Any]]:
        """Layer 1: Audit direct evidence of required components

        Enhanced with highly specific Bayesian priors for rare evidence items.
        Example: D2-Q4 risk matrix, D5-Q5 unwanted effects are rare in poor PDMs.
        """
        results = {}

        # Load highly specific priors for rare evidence types
        # D2-Q4: Risk matrices are rare in poor PDMs (high probative value as Smoking Gun)
        rare_evidence_priors = {
            'risk_matrix': {
                'prior_alpha': 1.5,  # Low alpha = rare occurrence
                'prior_beta': 12.0,  # High beta = high failure rate when absent
                'keywords': ['matriz de riesgo', 'anÃ¡lisis de riesgo', 'gestiÃ³n de riesgo', 'riesgos identificados']
            },
            'unwanted_effects': {
                'prior_alpha': 1.8,  # D5-Q5: Effects analysis is also rare
                'prior_beta': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "unwanted_effects_prior_beta", 10.5),
                'keywords': ['efectos no deseados', 'efectos adversos', 'impactos negativos',
                             'consecuencias no previstas']
            },
            'theory_of_change': {
                'prior_alpha': 1.2,
                'prior_beta': 15.0,
                'keywords': ['teorÃ-a de cambio', 'teorÃ-a del cambio', 'cadena causal', 'modelo lÃ³gico']
            }
        }

        for node_id, node in nodes.items():
```

```
        omissions = []
        omission_probs = {}
        rare_evidence_found = {}

        # Check for rare, high-value evidence in node text
        node_text_lower = node.text.lower()
        for evidence_type, prior_config in rare_evidence_priors.items():
            if any(kw in node_text_lower for kw in prior_config['keywords']):
                # Rare evidence found! Strong Smoking Gun
                rare_evidence_found[evidence_type] = {
                    'prior_alpha': prior_config['prior_alpha'],
                    'prior_beta': prior_config['prior_beta'],
                    'posterior_strength': prior_config['prior_alpha'] / (
                                prior_config['prior_alpha'] + prior_config['prior_beta'])
                }
                self.logger.info(f"Rare evidence '{evidence_type}' found in {node_id} - Strong Smoking Gun!")

        # Check baseline
        if not node.baseline or str(node.baseline).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
            p_failure_given_omission = 1.0 - historical_data.get('baseline_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "baseline_presence_success_rate", 0.89))
            omissions.append('baseline')
            omission_probs['baseline'] = p_failure_given_omission

        # Check target
        if not node.target or str(node.target).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
            p_failure_given_omission = 1.0 - historical_data.get('target_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "target_presence_success_rate", 0.92))
            omissions.append('target')
            omission_probs['target'] = p_failure_given_omission

        # Check entity
        if not node.responsible_entity:
            p_failure_given_omission = 1.0 - historical_data.get('entity_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "entity_presence_success_rate", 0.94))
            omissions.append('entity')
            omission_probs['entity'] = p_failure_given_omission

        # Check budget
        if not node.financial_allocation:
            p_failure_given_omission = 1.0 - historical_data.get('budget_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "budget_presence_success_rate", 0.78))
            omissions.append('budget')
            omission_probs['budget'] = p_failure_given_omission

        # Check mechanism
        if not node.entity_activity:
            p_failure_given_omission = 1.0 - historical_data.get('mechanism_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "mechanism_presence_success_rate", 0.65))
            omissions.append('mechanism')
            omission_probs['mechanism'] = p_failure_given_omission

        # Determine severity
        severity = 'none'
        if omission_probs:
            max_failure_prob = max(omission_probs.values())
```

```
                if max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "critical_threshold", 0.15):
                    severity = ’critical’
                elif max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "high_threshold", 0.10):
                    severity = ’high’
                elif max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "medium_threshold", 0.05):
                    severity = ’medium’
                else:
                    severity = ’low’

            results[node_id] = {
                ’omissions’: omissions,
                ’omission_probabilities’: omission_probs,
                ’omission_severity’: severity,
                ’node_type’: node.type,
                ’rare_evidence_found’: rare_evidence_found  # Add rare evidence to results
            }

        return results

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications")
    def _audit_causal_implications(self, nodes: dict[str, MetaNode],
                                   graph: nx.DiGraph,
                                   direct_evidence: dict[str, dict[str, Any]]) -> dict[str, dict[str, Any]]:
        """Layer 2: Audit causal implications of omissions"""
        implications = {}

        for node_id, node in nodes.items():
            node_omissions = direct_evidence[node_id][’omissions’]
            causal_effects = {}

            # If baseline is missing
            if ’baseline’ in node_omissions:
                # P(target_miscalibrated | missing_baseline)
                causal_effects[’target_miscalibration’] = {
                    ’probability’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "target_miscalibration_prob", 0.73),
                    ’description’: ’Sin lÃ-nea base, la meta probablemente estÃ¡ mal calibrada’
                }

            # If entity and high budget are missing
            if ’entity’ in node_omissions and node.financial_allocation and node.financial_allocation > 1000000:
                causal_effects[’implementation_failure’] = {
                    ’probability’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "implementation_failure_high_budget_prob", 0.89),
                    ’description’: ’Alto presupuesto sin entidad responsable indica alto riesgo de falla’
                }
            elif ’entity’ in node_omissions:
                causal_effects[’implementation_failure’] = {
                    ’probability’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "implementation_failure_prob", 0.65),
                    ’description’: ’Sin entidad responsable, la implementaciÃ³n es incierta’
                }

```

```
            # If mechanism is missing
            if 'mechanism' in node_omissions:
                causal_effects['unclear_pathway'] = {
                    'probability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "unclear_pathway_prob", 0.70),
                    'description': 'Sin mecanismo definido, la vÃ-a causal es opaca'
                }

            # Check downstream effects
            successors = list(graph.successors(node_id)) if graph.has_node(node_id) else []
            if node_omissions and successors:
                causal_effects['cascade_risk'] = {
                    'probability': min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "cascade_risk_max_prob", 0.95), ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "cascade_risk_base_prob", 0.4) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "cascade_risk_per_omission_prob", 0.1) * len(node_omissions)),
                    'affected_nodes': successors,
                    'description': f'Omisiones pueden afectar {len(successors)} nodos dependientes'
                }

            implications[node_id] = {
                'causal_effects': causal_effects,
                'total_risk': sum(e['probability'] for e in causal_effects.values()) / max(len(causal_effects), 1)
            }

        return implications

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk")
    def _audit_systemic_risk(self, nodes: dict[str, MetaNode],
                             graph: nx.DiGraph,
                             direct_evidence: dict[str, dict[str, Any]],
                             causal_implications: dict[str, dict[str, Any]],
                             pdet_alignment: float | None = None) -> dict[str, Any]:
        """
        AUDIT POINT 2.3: Policy Alignment Dual Constraint
        Layer 3: Calculate systemic risk from accumulated omissions

        Harmonic Front 3 - Enhancement 1: Alignment and Systemic Risk Linkage
        Incorporates Policy Alignment scores (PND, ODS, RRI) as variable in systemic risk.

        For D5-Q4 (Riesgos SistÃ©micos) and D4-Q5 (AlineaciÃ³n):
        - If pdet_alignment â\211¤ 0.60), applies 1.2Ã\227 multiplier to risk_score
        - Excelente on D5-Q4 requires risk_score < 0.10)

        Implements dual constraints integrating macro-micro causality per Lieberman 2015.
        """

        # Identify critical nodes (high centrality)
        if graph.number_of_nodes() > 0:
            try:
                centrality = nx.betweenness_centrality(graph)
            except (nx.NetworkXError, ZeroDivisionError, Exception) as e:
                logging.warning(f"Failed to calculate betweenness centrality: {e}. Using default values.")
                centrality = dict.fromkeys(graph.nodes(), 0.5)
        else:
            centrality = {}
```

```
        # Calculate P(cascade_failure | omission_set)
        critical_omissions = []
        for node_id, evidence in direct_evidence.items():
            if evidence['omission_severity'] in ['critical', 'high']:
                node_centrality = centrality.get(node_id, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "default_centrality", 0.5))
                critical_omissions.append({
                    'node_id': node_id,
                    'severity': evidence['omission_severity'],
                    'centrality': node_centrality,
                    'omissions': evidence['omissions']
                })

        # Calculate systemic risk
        if critical_omissions:
            # Weighted by centrality
            risk_score = sum(
                (ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "critical_severity_multiplier", 1.0) if om['severity'] == 'critical' else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "high_severity_multiplier", 0.7)) * (om['centrality'] + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "centrality_bonus", 0.1))
                for om in critical_omissions
            ) / len(nodes)
        else:
            risk_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_score", 0.0) # Refactored

        # AUDIT POINT 2.3: Policy Alignment Dual Constraint
        # If pdet_alignment â\211¤ ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "alignment_threshold", 0.60), apply 1.2Ã\227 multiplier to risk_score
        # This enforces integration between D4-Q5 (AlineaciÃ³n) and D5-Q4 (Riesgos SistÃ©micos)
        alignment_penalty_applied = False
        alignment_threshold = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "alignment_threshold", 0.6) # Refactored
        alignment_multiplier = 1.2

        if pdet_alignment is not None and pdet_alignment <= alignment_threshold:
            original_risk = risk_score
            risk_score = risk_score * alignment_multiplier
            alignment_penalty_applied = True
            self.logger.warning(
                f"ALIGNMENT PENALTY (D5-Q4): pdet_alignment={pdet_alignment:.2f} â\211¤ {alignment_threshold}, "
                f"risk_score escalated from {original_risk:.3f} to {risk_score:.3f} "
                f"(multiplier: {alignment_multiplier}Ã\227). Dual constraint per Lieberman 2015."
            )

        # Calculate P(success | current_state)
        total_omissions = sum(len(e['omissions']) for e in direct_evidence.values())
        total_possible = len(nodes) * 5  # 5 key attributes per node
        completeness = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "completeness_factor", 1.0) - (total_omissions / max(total_possible, 1))

        # Success probability (simplified Bayesian)
        base_success_rate = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "base_success_rate", 0.7) # Refactored
```

```
        success_probability = base_success_rate * completeness

        # D5-Q4 quality criteria check (AUDIT POINT 2.3)
        # Excellent requires risk_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_excellent", 0.10) (matching ODS benchmarks per UN 2020)
        d5_q4_quality = ’insuficiente’
        risk_threshold_excellent = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_excellent", 0.1) # Refactored
        risk_threshold_good = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_good", 0.2) # Refactored
        risk_threshold_acceptable = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_acceptable", 0.35) # Refactored

        if risk_score < risk_threshold_excellent:
            d5_q4_quality = ’excelente’
        elif risk_score < risk_threshold_good:
            d5_q4_quality = ’bueno’
        elif risk_score < risk_threshold_acceptable:
            d5_q4_quality = ’aceptable’

        # Flag if alignment is causing quality failure
        alignment_causing_failure = (
                alignment_penalty_applied and
                original_risk < risk_threshold_excellent and
                risk_score >= risk_threshold_excellent
        )

        return {
            ’risk_score’: min(1.0, risk_score),
            ’success_probability’: success_probability,
            ’critical_omissions’: critical_omissions,
            ’completeness’: completeness,
            ’total_omissions’: total_omissions,
            ’pdet_alignment’: pdet_alignment,
            ’alignment_penalty_applied’: alignment_penalty_applied,
            ’alignment_threshold’: alignment_threshold,
            ’alignment_multiplier’: alignment_multiplier,
            ’alignment_causing_failure’: alignment_causing_failure,
            ’d5_q4_quality’: d5_q4_quality,
            ’d4_q5_alignment_score’: pdet_alignment,
            ’risk_thresholds’: {
                ’excellent’: risk_threshold_excellent,
                ’good’: risk_threshold_good,
                ’acceptable’: risk_threshold_acceptable
            }
        }

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._generate_optimal_remediations")
    def _generate_optimal_remediations(self,
                                       direct_evidence: dict[str, dict[str, Any]],
                                       causal_implications: dict[str, dict[str, Any]],
                                       systemic_risk: dict[str, Any]) -> list[dict[str, Any]]:
        """Generate prioritized remediation recommendations"""
        remediations = []

        # Calculate expected value of information for each remediation
```

```
        for node_id, evidence in direct_evidence.items():
            if not evidence[’omissions’]:
                continue

            for omission in evidence[’omissions’]:
                # Estimate impact
                omission_prob = evidence[’omission_probabilities’].get(omission, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._generate_optimal_remediations", "default_omission_prob", 0.1))
                causal_risk = causal_implications[node_id][’total_risk’]

                # Expected value = P(failure_avoided) * Impact
                expected_value = omission_prob * (1 + causal_risk)

                # Effort estimate (simplified)
                effort_map = {
                    ’baseline’: 3,  # Moderate effort to research
                    ’target’: 2,  # Low effort to define
                    ’entity’: 2,  # Low effort to assign
                    ’budget’: 4,  # Higher effort to allocate
                    ’mechanism’: 5  # Highest effort to design
                }
                effort = effort_map.get(omission, 3)

                # Priority = Expected Value / Effort
                priority = expected_value / effort

                remediations.append({
                    ’node_id’: node_id,
                    ’omission’: omission,
                    ’severity’: evidence[’omission_severity’],
                    ’expected_value’: expected_value,
                    ’effort’: effort,
                    ’priority’: priority,
                    ’recommendation’: self._get_remediation_text(omission, node_id)
                })

        # Sort by priority (descending)
        remediations.sort(key=lambda x: x[’priority’], reverse=True)

        return remediations

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_remediation_text")
    def _get_remediation_text(self, omission: str, node_id: str) -> str:
        """Get specific remediation text for an omission"""
        texts = {
            ’baseline’: f"Definir lÃ-nea base cuantitativa para {node_id} basada en diagnÃ³stico actual",
            ’target’: f"Especificar meta cuantitativa alcanzable para {node_id} con horizonte temporal",
            ’entity’: f"Asignar entidad responsable clara para la ejecuciÃ³n de {node_id}",
            ’budget’: f"Asignar recursos presupuestarios especÃ-ficos a {node_id}",
            ’mechanism’: f"Documentar mecanismo causal (Entidad-Actividad) para {node_id}"
        }
        return texts.get(omission, f"Completar {omission} para {node_id}")

    @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._perform_counterfactual_budget_check")
    def _perform_counterfactual_budget_check(self, nodes: dict[str, MetaNode],
                                             graph: nx.DiGraph) -> dict[str, Any]:
```

```
        """
        Perform counterfactual budget check for operationalization audit.

        This method evaluates whether removing budget allocation would prevent
        goal execution, helping identify necessary vs. superfluous allocations.

        Args:
            nodes: Dictionary of meta nodes
            graph: Causal graph

        Returns:
            Dictionary with counterfactual analysis results
        """
        results = {
            ’nodes_analyzed’: 0,
            ’budget_necessary’: [],
            ’budget_optional’: [],
            ’unallocated’: []
        }

        for node_id, node in nodes.items():
            results[’nodes_analyzed’] += 1

            has_budget = node.financial_allocation and node.financial_allocation > 0
            has_mechanism = node.entity_activity is not None
            has_dependencies = len(list(graph.successors(node_id))) > 0 if graph.has_node(node_id) else False

            if not has_budget:
                results[’unallocated’].append(node_id)
            elif has_mechanism and has_dependencies:
                # Budget seems necessary for execution
                results[’budget_necessary’].append(node_id)
            else:
                # Budget may be optional or disconnected
                results[’budget_optional’].append(node_id)

        return results

class BayesianMechanismInference:
    """
    AGUJA II: El Modelo Generativo de Mecanismos
    Hierarchical Bayesian model for causal mechanism inference

    F1.2 ARCHITECTURAL REFACTORING:
    This class now integrates with refactored Bayesian engine components:
    - BayesianPriorBuilder: Construye priors adaptativos (AGUJA I)
    - BayesianSamplingEngine: Ejecuta MCMC sampling (AGUJA II)
    - NecessitySufficiencyTester: Ejecuta Hoop Tests (AGUJA III)

    The refactored components provide:
    - Crystal-clear separation of concerns
    - Trivial unit testing
    - Explicit compliance with Fronts B and C

    Legacy methods are preserved for backward compatibility.
    """
```

```
    def __init__(self, config: ConfigLoader, nlp_model: spacy.Language, **kwargs) -> None:
        """
        Initialize Bayesian Mechanism Inference engine.

        Args:
            config: Configuration loader instance
            nlp_model: spaCy NLP model for text processing
            **kwargs: Accepts additional keyword arguments for backward compatibility.
                     Unexpected arguments (e.g., 'causal_hierarchy') are logged and ignored.

        Note:
            This function signature has been made defensive to handle unexpected
            keyword arguments that may be passed due to interface drift.
        """
        # Log warning if unexpected kwargs are passed
        if kwargs:
            logging.getLogger(__name__).warning(
                f"BayesianMechanismInference.__init__ received unexpected keyword arguments: {list(kwargs.keys())}. "
                "These will be ignored. Expected signature: __init__(self, config: ConfigLoader, nlp_model: spacy.Language)"
            )

        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.nlp = nlp_model

        # F1.2: Initialize refactored Bayesian engine adapter if available
        if REFACTORED_BAYESIAN_AVAILABLE:
            try:
                self.bayesian_adapter = BayesianEngineAdapter(config, nlp_model)
                if self.bayesian_adapter.is_available():
                    self.logger.info("â\234\223 Usando motor Bayesiano refactorizado (F1.2)")
                    self._log_refactored_components()
                else:
                    self.bayesian_adapter = None
            except Exception as e:
                self.logger.warning(f"Error inicializando motor refactorizado: {e}")
                self.bayesian_adapter = None
        else:
            self.bayesian_adapter = None

        # Load mechanism type hyperpriors from configuration (externalized)
        self.mechanism_type_priors = {
            'administrativo': self.config.get_mechanism_prior('administrativo'),
            'tecnico': self.config.get_mechanism_prior('tecnico'),
            'financiero': self.config.get_mechanism_prior('financiero'),
            'politico': self.config.get_mechanism_prior('politico'),
            'mixto': self.config.get_mechanism_prior('mixto')
        }

        # Typical activity sequences by mechanism type
        # These could also be externalized if needed for domain-specific customization
        self.mechanism_sequences = {
            'administrativo': ['planificar', 'coordinar', 'gestionar', 'supervisar'],
            'tecnico': ['diagnosticar', 'diseÃ±ar', 'implementar', 'evaluar'],
            'financiero': ['asignar', 'ejecutar', 'auditar', 'reportar'],
```

```
            'politico': ['concertar', 'negociar', 'aprobar', 'promulgar']
        }

        # Track inferred mechanisms
        self.inferred_mechanisms: dict[str, dict[str, Any]] = {}

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._log_refactored_components")
    def _log_refactored_components(self) -> None:
        """Log status of refactored Bayesian components (F1.2)"""
        if self.bayesian_adapter:
            status = self.bayesian_adapter.get_component_status()
            self.logger.info("  - BayesianPriorBuilder: " +
                             ("â\234\223" if status['prior_builder_ready'] else "â\234\227"))
            self.logger.info("  - BayesianSamplingEngine: " +
                             ("â\234\223" if status['sampling_engine_ready'] else "â\234\227"))
            self.logger.info("  - NecessitySufficiencyTester: " +
                             ("â\234\223" if status['necessity_tester_ready'] else "â\234\227"))

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms")
    def infer_mechanisms(self, nodes: dict[str, MetaNode],
                         text: str) -> dict[str, dict[str, Any]]:
        """
        Infer latent causal mechanisms using hierarchical Bayesian modeling

        HARMONIC FRONT 4 ENHANCEMENT:
        - Tracks mean mechanism_type uncertainty for quality criteria
        - Reports uncertainty reduction metrics
        """
        self.logger.info("Iniciando inferencia Bayesiana de mecanismos...")

        # Focus on 'producto' nodes which should have mechanisms
        product_nodes = {nid: n for nid, n in nodes.items() if n.type == 'producto'}

        # Track uncertainties for mean calculation
        mechanism_uncertainties = []

        for node_id, node in product_nodes.items():
            mechanism = self._infer_single_mechanism(node, text, nodes)
            self.inferred_mechanisms[node_id] = mechanism

            # Track mechanism type uncertainty for quality criteria
            if 'uncertainty' in mechanism:
                mech_type_uncertainty = mechanism['uncertainty'].get('mechanism_type', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms", "default_uncertainty", 1.0))
                mechanism_uncertainties.append(mech_type_uncertainty)

        # Calculate mean mechanism uncertainty for Harmonic Front 4 quality criteria
        mean_mech_uncertainty = (
            np.mean(mechanism_uncertainties) if mechanism_uncertainties else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms", "default_mean_uncertainty", 1.0)
        )

        self.logger.info(f"Mecanismos inferidos: {len(self.inferred_mechanisms)}")
        self.logger.info(f"Mean mechanism_type uncertainty: {mean_mech_uncertainty:.4f}")

        # Store for reporting
```

```
        self._mean_mechanism_uncertainty = mean_mech_uncertainty

        return self.inferred_mechanisms

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_single_mechanism")
    def _infer_single_mechanism(self, node: MetaNode, text: str,
                                all_nodes: dict[str, MetaNode]) -> dict[str, Any]:
        """Infer mechanism for a single product node"""
        # Extract observations from text
        observations = self._extract_observations(node, text)

        # Level 3: Sample mechanism type from hyperprior
        mechanism_type_posterior = self._infer_mechanism_type(observations)

        # Level 2: Infer activity sequence given mechanism type
        sequence_posterior = self._infer_activity_sequence(
            observations, mechanism_type_posterior
        )

        # Level 1: Calculate coherence factor
        coherence_score = self._calculate_coherence_factor(
            node, observations, all_nodes
        )

        # Validation tests
        sufficiency = self._test_sufficiency(node, observations)
        necessity = self._test_necessity(node, observations)

        # Quantify uncertainty
        uncertainty = self._quantify_uncertainty(
            mechanism_type_posterior, sequence_posterior, coherence_score
        )

        # Detect gaps
        gaps = self._detect_gaps(node, observations, uncertainty)

        return {
            'mechanism_type': mechanism_type_posterior,
            'activity_sequence': sequence_posterior,
            'coherence_score': coherence_score,
            'sufficiency_test': sufficiency,
            'necessity_test': necessity,
            'uncertainty': uncertainty,
            'gaps': gaps,
            'observations': observations
        }

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._extract_observations")
    def _extract_observations(self, node: MetaNode, text: str) -> dict[str, Any]:
        """Extract textual observations related to the mechanism"""
        # Find node context in text
        node_pattern = re.escape(node.id)
        matches = list(re.finditer(node_pattern, text, re.IGNORECASE))

        observations = {
            'entity_activity': None,
```

```
            'verbs': [],
            'entities': [],
            'budget': node.financial_allocation,
            'context_snippets': []
        }

        if node.entity_activity:
            observations['entity_activity'] = {
                'entity': node.entity_activity.entity,
                'activity': node.entity_activity.activity,
                'verb_lemma': node.entity_activity.verb_lemma
            }

        # Extract context around node mentions
        for match in matches[:3]:  # Limit to first 3 occurrences
            start = max(0, match.start() - 300)
            end = min(len(text), match.end() + 300)
            context = text[start:end]

            # Process with spaCy
            doc = self.nlp(context)

            # Extract verbs
            verbs = [token.lemma_ for token in doc if token.pos_ == 'VERB']
            observations['verbs'].extend(verbs)

            # Extract entities
            entities = [ent.text for ent in doc.ents if ent.label_ in ['ORG', 'PER']]
            observations['entities'].extend(entities)

            observations['context_snippets'].append(context[:200])

        return observations

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_mechanism_type")
    def _infer_mechanism_type(self, observations: dict[str, Any]) -> dict[str, float]:
        """Infer mechanism type using Bayesian updating"""
        # Start with hyperprior
        posterior = dict(self.mechanism_type_priors)

        # Get Laplace smoothing parameter from configuration
        laplace_smooth = self.config.get_bayesian_threshold('laplace_smoothing')

        # Update based on observed verbs
        observed_verbs = set(observations.get('verbs', []))

        if observed_verbs:
            for mech_type, typical_verbs in self.mechanism_sequences.items():
                # Count overlap
                overlap = len(observed_verbs.intersection(set(typical_verbs)))
                total = len(typical_verbs)

                if total > 0:
                    # Likelihood: proportion of typical verbs observed with Laplace smoothing
                    likelihood = (overlap + laplace_smooth) / (total + 2 * laplace_smooth)

```

```
                    # Bayesian update
                    posterior[mech_type] *= likelihood

        # Update based on entity-activity
        if observations.get('entity_activity'):
            verb = observations['entity_activity'].get('verb_lemma', '')
            for mech_type, typical_verbs in self.mechanism_sequences.items():
                if verb in typical_verbs:
                    posterior[mech_type] *= 1.5

        # Normalize
        total = sum(posterior.values())
        if total > 0:
            posterior = {k: v / total for k, v in posterior.items()}

        return posterior

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_activity_sequence")
    def _infer_activity_sequence(self, observations: dict[str, Any],
                                 mechanism_type_posterior: dict[str, float]) -> dict[str, Any]:
        """Infer activity sequence parameters"""
        # Get most likely mechanism type
        best_type = max(mechanism_type_posterior.items(), key=lambda x: x[1])[0]
        expected_sequence = self.mechanism_sequences.get(best_type, [])

        observed_verbs = observations.get('verbs', [])

        # Calculate transition probabilities (simplified Markov chain)
        transitions = {}
        for i in range(len(expected_sequence) - 1):
            current = expected_sequence[i]
            next_verb = expected_sequence[i + 1]

            # Check if transition is observed
            if current in observed_verbs and next_verb in observed_verbs:
                transitions[(current, next_verb)] = 0.85
            else:
                transitions[(current, next_verb)] = 0.40

        return {
            'expected_sequence': expected_sequence,
            'observed_verbs': observed_verbs,
            'transition_probabilities': transitions,
            'sequence_completeness': len(set(observed_verbs) & set(expected_sequence)) / max(len(expected_sequence), 1)
        }

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_coherence_factor")
    def _calculate_coherence_factor(self, node: MetaNode,
                                    observations: dict[str, Any],
                                    all_nodes: dict[str, MetaNode]) -> float:
        """Calculate mechanism coherence score"""
        coherence = 0.0 # Refactored
        weights = []

        # Factor 1: Entity-Activity presence
        if observations.get('entity_activity'):
```

```
            coherence += 0.30
            weights.append(0.30)

        # Factor 2: Budget consistency
        if observations.get('budget'):
            coherence += 0.20
            weights.append(0.20)

        # Factor 3: Verb sequence completeness
        seq_info = observations.get('verbs', [])
        if seq_info:
            verb_score = min(len(seq_info) / 4.0, 1.0)  # Expect ~4 verbs
            coherence += verb_score * 0.25
            weights.append(0.25)

        # Factor 4: Entity presence
        if observations.get('entities'):
            coherence += 0.15
            weights.append(0.15)

        # Factor 5: Context richness
        snippets = observations.get('context_snippets', [])
        if snippets:
            coherence += 0.10
            weights.append(0.10)

        # Normalize by actual weights used
        if weights:
            coherence = coherence / sum(weights) if sum(weights) > 0 else 0.0

        return coherence

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._test_sufficiency")
    def _test_sufficiency(self, node: MetaNode,
                          observations: dict[str, Any]) -> dict[str, Any]:
        """Test if mechanism is sufficient to produce the outcome"""
        # Check if entity has capability
        has_entity = observations.get('entity_activity') is not None

        # Check if activities are present
        has_activities = len(observations.get('verbs', [])) >= 2

        # Check if resources are allocated
        has_resources = observations.get('budget') is not None

        sufficiency_score = (
                (0.4 if has_entity else 0.0) +
                (0.4 if has_activities else 0.0) +
                (0.2 if has_resources else 0.0)
        )

        return {
            'score': sufficiency_score,
            'is_sufficient': sufficiency_score >= 0.6,
            'components': {
                'entity': has_entity,
```

```
                'activities': has_activities,
                'resources': has_resources
            }
        }

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._test_necessity")
    def _test_necessity(self, node: MetaNode,
                        observations: dict[str, Any]) -> dict[str, Any]:
        """
        AUDIT POINT 2.2: Mechanism Necessity Hoop Test

        Test if mechanism is necessary by checking documented components:
        - Entity (responsable)
        - Activity (verb lemma sequence)
        - Budget (presupuesto asignado)

        Implements Beach 2017 Hoop Tests for necessity verification.
        Per Falleti & Lynch 2009, Bayesian-deterministic hybrid boosts mechanism depth.

        Returns:
            Dict with 'is_necessary', 'missing_components', and remediation text
        """
        # F1.2: Use refactored NecessitySufficiencyTester if available
        if self.bayesian_adapter and self.bayesian_adapter.necessity_tester:
            try:
                return self.bayesian_adapter.test_necessity_from_observations(
                    node.id,
                    observations
                )
            except Exception as e:
                self.logger.warning(f"Error en tester refactorizado: {e}, usando legacy")

        # AUDIT POINT 2.2: Enhanced necessity test with documented components
        missing_components = []

        # 1. Check Entity documentation
        entities = observations.get('entities', [])
        entity_activity = observations.get('entity_activity')

        if not entity_activity or not entity_activity.get('entity'):
            missing_components.append('entity')
        else:
            # Verify unique entity (not multiple conflicting entities)
            unique_entity = len(set(entities)) == 1 if entities else False
            if not unique_entity and len(entities) > 1:
                missing_components.append('unique_entity')

        # 2. Check Activity documentation (verb lemma sequence)
        verbs = observations.get('verbs', [])
        if not verbs or len(verbs) < 1:
            missing_components.append('activity')
        else:
            # Check for specific action verbs (not just generic ones)
            specific_verbs = [v for v in verbs if v in [
                'implementar', 'ejecutar', 'realizar', 'desarrollar',
                'construir', 'diseÃ±ar', 'planificar', 'coordinar',
```

```
                'gestionar', 'supervisar', 'controlar', 'auditar'
            ]]
            if not specific_verbs:
                missing_components.append('specific_activity')

        # 3. Check Budget documentation
        budget = observations.get('budget')
        if budget is None or budget <= 0:
            missing_components.append('budget')

        # Calculate necessity score
        # All three components must be present for necessity=True
        is_necessary = len(missing_components) == 0

        # Calculate partial score for reporting
        max_components = 3  # entity, activity, budget
        present_components = max_components - len(
            [c for c in missing_components if c in ['entity', 'activity', 'budget']])
        necessity_score = present_components / max_components

        result = {
            'score': necessity_score,
            'is_necessary': is_necessary,
            'missing_components': missing_components,
            'alternatives_likely': not is_necessary,
            'hoop_test_passed': is_necessary
        }

        # Add remediation text if test fails
        if not is_necessary:
            result['remediation'] = self._generate_necessity_remediation(node.id, missing_components)

        return result

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._generate_necessity_remediation")
    def _generate_necessity_remediation(self, node_id: str, missing_components: list[str]) -> str:
        """Generate remediation text for failed necessity test"""
        component_descriptions = {
            'entity': 'entidad responsable claramente identificada',
            'unique_entity': 'una Ãºnica entidad responsable (mÃºltiples entidades detectadas)',
            'activity': 'secuencia de actividades documentada',
            'specific_activity': 'actividades especÃficas (no genÃ©ricas)',
            'budget': 'presupuesto asignado y cuantificado'
        }

        missing_desc = ', '.join([component_descriptions.get(c, c) for c in missing_components])

        return (
            f"Mecanismo para {node_id} falla Hoop Test de necesidad (D6-Q2). "
            f"Componentes faltantes: {missing_desc}. "
            f"Se requiere documentar estos componentes necesarios para validar "
            f"la cadena causal segÃºn Beach 2017."
        )

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty")
    def _quantify_uncertainty(self, mechanism_type_posterior: dict[str, float],
```

```
                              sequence_posterior: dict[str, Any],
                              coherence_score: float) -> dict[str, float]:
        """Quantify epistemic uncertainty"""
        # Entropy of mechanism type distribution
        mech_probs = list(mechanism_type_posterior.values())
        if mech_probs:
            mech_entropy = -sum(p * np.log(p + 1e-10) for p in mech_probs if p > 0)
            max_entropy = np.log(len(mech_probs))
            mech_uncertainty = mech_entropy / max_entropy if max_entropy > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_mech_uncertainty", 1.0)
        else:
            mech_uncertainty = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_mech_uncertainty", 1.0) # Refactored

        # Sequence completeness uncertainty
        seq_completeness = sequence_posterior.get('sequence_completeness', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_seq_completeness", 0.0))
        seq_uncertainty = 1.0 - seq_completeness

        # Coherence uncertainty
        coherence_uncertainty = 1.0 - coherence_score

        # Combined uncertainty
        total_uncertainty = (
                mech_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "mech_uncertainty_weight", 0.4) +
                seq_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "seq_uncertainty_weight", 0.3) +
                coherence_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "coherence_uncertainty_weight", 0.3)
        )

        return {
            'total': total_uncertainty,
            'mechanism_type': mech_uncertainty,
            'sequence': seq_uncertainty,
            'coherence': coherence_uncertainty
        }

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._detect_gaps")
    def _detect_gaps(self, node: MetaNode, observations: dict[str, Any],
                     uncertainty: dict[str, float]) -> list[dict[str, str]]:
        """Detect documentation gaps based on uncertainty"""
        gaps = []

        # High total uncertainty
        if uncertainty['total'] > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._detect_gaps", "high_uncertainty_threshold", 0.6):
            gaps.append({
                'type': 'high_uncertainty',
                'severity': 'high',
                'message': f"Mecanismo para {node.id} tiene alta incertidumbre ({uncertainty['total']:.2f})",
                'suggestion': "Se requiere mÃ¡s documentaciÃ³n sobre el mecanismo causal"
            })

        # Missing entity
```

```
        if not observations.get('entity_activity'):
            gaps.append({
                'type': 'missing_entity',
                'severity': 'high',
                'message': f"No se especifica entidad responsable para {node.id}",
                'suggestion': "Especificar quÃ© entidad ejecutarÃ¡ las actividades"
            })

        # Insufficient activities
        if len(observations.get('verbs', [])) < 2:
            gaps.append({
                'type': 'insufficient_activities',
                'severity': 'medium',
                'message': f"Pocas actividades documentadas para {node.id}",
                'suggestion': "Detallar las actividades necesarias para lograr el producto"
            })

        # Missing budget
        if not observations.get('budget'):
            gaps.append({
                'type': 'missing_budget',
                'severity': 'medium',
                'message': f"Sin asignaciÃ³n presupuestaria para {node.id}",
                'suggestion': "Asignar recursos financieros al producto"
            })

        return gaps

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._aggregate_bayesian_confidence")
    def _aggregate_bayesian_confidence(self, confidences: list[float]) -> float:
        """
        Aggregate multiple Bayesian confidence values.

        Args:
            confidences: List of confidence values to aggregate

        Returns:
            Aggregated confidence value
        """
        if not confidences:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._aggregate_bayesian_confidence", "default_confidence", 
0.5)  # Default neutral confidence
        return float(np.mean(confidences))

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix")
    def _build_transition_matrix(self, mechanism_type: str) -> np.ndarray:
        """
        Build transition matrix for activity sequences.

        Args:
            mechanism_type: Type of mechanism

        Returns:
            Transition probability matrix
        """
        # Get typical sequence for this mechanism type
```

```
        sequence = self.mechanism_sequences.get(mechanism_type, ['planificar', 'ejecutar', 'evaluar'])
        n = len(sequence)

        # Create a simple sequential transition matrix
        matrix = np.zeros((n, n))
        for i in range(n - 1):
            matrix[i, i + 1] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "next_step_prob", 0.7)  # High probability of next step
            matrix[i, i] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "stay_prob", 0.2)  # Some probability of staying in same step
            if i < n - 2:
                matrix[i, i + 2] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "skip_prob", 0.1)  # Small probability of skipping
        matrix[n - 1, n - 1] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "absorbing_prob", 1.0)  # Final state is absorbing

        return matrix

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior")
    def _calculate_type_transition_prior(self, from_type: str, to_type: str) -> float:
        """
        Calculate prior probability of transitioning between mechanism types.

        Args:
            from_type: Source mechanism type
            to_type: Target mechanism type

        Returns:
            Prior probability of transition
        """
        # Same type has high probability
        if from_type == to_type:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "same_type_prob", 0.7)


        # Related types have medium probability
        related_pairs = [
            ('administrativo', 'politico'),
            ('tecnico', 'financiero'),
            ('financiero', 'administrativo'),
        ]
        if (from_type, to_type) in related_pairs or (to_type, from_type) in related_pairs:
            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "related_type_prob", 0.2)

        # Unrelated types have low probability
        return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "unrelated_type_prob", 0.1)

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type")
    def _classify_mechanism_type(self, observations: dict[str, Any]) -> str:
        """
        Classify mechanism type based on observations.

        Args:
            observations: Observed features
```

```

        Returns:
            Classified mechanism type
        """
        # Extract features
        verbs = observations.get('verbs', [])
        entities = observations.get('entities', [])
        budget = observations.get('budget')

        # Score each mechanism type
        scores = {}
        for mech_type, typical_verbs in self.mechanism_sequences.items():
            score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "score", 0.0) # Refactored
            # Count matching verbs
            for verb in verbs:
                if any(tv in verb.lower() for tv in typical_verbs):
                    score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "verb_match_bonus", 1.0)
            scores[mech_type] = score

        # Adjust for budget presence (indicates financial mechanism)
        if budget and budget > 0:
            scores['financiero'] = scores.get('financiero', 0) + 2.0

        # Adjust for political/administrative entities
        for entity in entities:
            entity_lower = entity.lower()
            if any(word in entity_lower for word in ['alcaldÃa', 'consejo', 'gobernaciÃ³n']):
                scores['politico'] = scores.get('politico', 0) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "political_entity_bonus", 1.0)
            if any(word in entity_lower for word in ['secretarÃa', 'direcciÃ³n', 'oficina']):
                scores['administrativo'] = scores.get('administrativo', 0) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "administrative_entity_bonus", 1.0)

        # Return type with highest score, or 'mixto' if tie
        if not scores or all(s == 0 for s in scores.values()):
            return 'mixto'

        max_score = max(scores.values())
        max_types = [t for t, s in scores.items() if s == max_score]

        if len(max_types) > 1:
            return 'mixto'
        return max_types[0]

class CausalInferenceSetup:
    """Prepare model for causal inference"""

    def __init__(self, config: ConfigLoader) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.goal_classification = config.get('lexicons.goal_classification', {})
        self.admin_keywords = config.get('lexicons.administrative_keywords', [])
        self.contextual_factors = config.get('lexicons.contextual_factors', [])

    @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.classify_goal_dynamics")
```

```
    def classify_goal_dynamics(self, nodes: dict[str, MetaNode]) -> None:
        """Classify dynamics for each goal"""
        for node in nodes.values():
            text_lower = node.text.lower()

            for keyword, dynamics in self.goal_classification.items():
                if keyword in text_lower:
                    node.dynamics = cast("DynamicsType", dynamics)
                    self.logger.debug(f"Meta {node.id} clasificada como {node.dynamics}")
                    break

    @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.assign_probative_value")
    def assign_probative_value(self, nodes: dict[str, MetaNode]) -> None:
        """Assign probative test types to nodes"""
        # Import INDICATOR_STRUCTURE from financiero_viabilidad_tablas
        try:
            from financiero_viabilidad_tablas import ColombianMunicipalContext
            indicator_structure = ColombianMunicipalContext.INDICATOR_STRUCTURE
        except ImportError:
            indicator_structure = {
                ’resultado’: [’lÃ-nea_base’, ’meta’, ’aÃ±o_base’, ’aÃ±o_meta’, ’fuente’, ’responsable’],
                ’producto’: [’indicador’, ’fÃ³rmula’, ’unidad_medida’, ’lÃ-nea_base’, ’meta’, ’periodicidad’],
                ’gestiÃ³n’: [’eficacia’, ’eficiencia’, ’economÃ-a’, ’costo_beneficio’]
            }

        for node in nodes.values():
            text_lower = node.text.lower()

            # Cross-reference with INDICATOR_STRUCTURE to classify critical requirements
            # as Hoop Tests or Smoking Guns
            indicator_structure.get(node.type, [])

            # Check if node has all critical DNP requirements (D3-Q1 indicators)
            has_linea_base = bool(
                node.baseline and str(node.baseline).upper() not in [’ND’, ’POR DEFINIR’, ’N/A’, ’NONE’])
            has_meta = bool(node.target and str(node.target).upper() not in [’ND’, ’POR DEFINIR’, ’N/A’, ’NONE’])
            has_fuente = ’fuente’ in text_lower or ’fuente de informaciÃ³n’ in text_lower

            # Perfect Hoop Test: Missing any critical requirement = total hypothesis failure
            # This applies to producto nodes with D3-Q1 indicators
            if node.type == ’producto’:
                if has_linea_base and has_meta and has_fuente:
                    # Perfect indicators trigger Hoop Test classification
                    node.test_type = ’hoop_test’
                    self.logger.debug(f"Meta {node.id} classified as hoop_test (perfect D3-Q1 compliance)")
                elif not has_linea_base or not has_meta:
                    # Missing critical requirements - still Hoop Test but will fail
                    node.test_type = ’hoop_test’
                    node.audit_flags.append(’hoop_test_failure’)
                    self.logger.warning(f"Meta {node.id} FAILS hoop_test (missing D3-Q1 critical fields)")
                else:
                    node.test_type = ’straw_in_wind’
            # Check for administrative/regulatory nature (Hoop Test)
            elif any(keyword in text_lower for keyword in self.admin_keywords):
                node.test_type = ’hoop_test’
            # Check for highly specific outcomes (Smoking Gun)
```

```
            elif node.type == 'resultado' and node.target and node.baseline:
                try:
                    float(str(node.target).replace(',', '').replace('%', ''))
                    # Smoking Gun: rare, highly specific evidence with strong inferential power
                    node.test_type = 'smoking_gun'
                except (ValueError, TypeError):
                    node.test_type = 'straw_in_wind'
            # Double decisive for critical impact goals
            elif node.type == 'impacto' and node.rigor_status == 'fuerte':
                node.test_type = 'doubly_decisive'
            else:
                node.test_type = 'straw_in_wind'

            self.logger.debug(f"Meta {node.id} asignada test type: {node.test_type}")

    @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.identify_failure_points")
    def identify_failure_points(self, graph: nx.DiGraph, text: str) -> set[str]:
        """Identify single points of failure in causal chain

        Harmonic Front 3 - Enhancement 2: Contextual Failure Point Detection
        Expands risk_pattern to explicitly include localized contextual factors from rubrics:
        - restricciones territoriales
        - patrones culturales machistas
        - limitaciÃ³n normativa

        For D6-Q5 (Enfoque Diferencial/Restricciones): Excelente requires â\211¥3 distinct
        contextual factors correctly mapped to nodes, satisfying enfoque_diferencial
        and analisis_contextual criteria.
        """
        failure_points = set()

        # Find nodes with high out-degree (many dependencies)
        for node_id in graph.nodes():
            out_degree = graph.out_degree(node_id)
            node_type = graph.nodes[node_id].get('type')

            if node_type == 'producto' and out_degree >= 3:
                failure_points.add(node_id)
                self.logger.warning(f"Punto Ãºnico de falla identificado: {node_id} "
                                    f"(grado de salida: {out_degree})")

        # HARMONIC FRONT 3 - Enhancement 2: Expand contextual factors
        # Add specific rubric factors for D6-Q5 compliance
        extended_contextual_factors = list(self.contextual_factors) + [
            'restricciones territoriales',
            'restricciÃ³n territorial',
            'limitaciÃ³n territorial',
            'patrones culturales machistas',
            'machismo',
            'inequidad de gÃ©nero',
            'violencia de gÃ©nero',
            'limitaciÃ³n normativa',
            'limitaciÃ³n legal',
            'restricciÃ³n legal',
            'barrera institucional',
            'restricciÃ³n presupuestal',
```

```
            'ausencia de capacidad tÃ©cnica',
            'baja capacidad institucional',
            'conflicto armado',
            'desplazamiento forzado',
            'poblaciÃ³n dispersa',
            'ruralidad dispersa',
            'acceso vial limitado',
            'conectividad deficiente'
        ]

        # Extract contextual risks from text
        risk_pattern = '|'.join(re.escape(factor) for factor in extended_contextual_factors)
        risk_regex = re.compile(rf'\b({risk_pattern})\b', re.IGNORECASE)

        # Track distinct contextual factors for D6-Q5 quality criteria
        contextual_factors_detected = set()
        node_contextual_map = defaultdict(set)

        # Find risk mentions and associate with nodes
        for match in risk_regex.finditer(text):
            risk_text = match.group()
            contextual_factors_detected.add(risk_text.lower())

            context_start = max(0, match.start() - 200)
            context_end = min(len(text), match.end() + 200)
            context = text[context_start:context_end]

            # Try to find node mentions in risk context
            for node_id in graph.nodes():
                if node_id in context:
                    failure_points.add(node_id)
                    if 'contextual_risks' not in graph.nodes[node_id]:
                        graph.nodes[node_id]['contextual_risks'] = []
                    graph.nodes[node_id]['contextual_risks'].append(risk_text)
                    node_contextual_map[node_id].add(risk_text.lower())

        # D6-Q5 quality criteria assessment
        distinct_factors_count = len(contextual_factors_detected)
        d6_q5_quality = 'insuficiente'
        if distinct_factors_count >= 3:
            d6_q5_quality = 'excelente'
        elif distinct_factors_count >= 2:
            d6_q5_quality = 'bueno'
        elif distinct_factors_count >= 1:
            d6_q5_quality = 'aceptable'

        # Store D6-Q5 metrics in graph attributes
        graph.graph['d6_q5_contextual_factors'] = list(contextual_factors_detected)
        graph.graph['d6_q5_distinct_count'] = distinct_factors_count
        graph.graph['d6_q5_quality'] = d6_q5_quality
        graph.graph['d6_q5_node_mapping'] = dict(node_contextual_map)

        self.logger.info(f"Puntos de falla identificados: {len(failure_points)}")
        self.logger.info(
            f"D6-Q5: {distinct_factors_count} factores contextuales distintos detectados - {d6_q5_quality}")

```

```
        return failure_points

    @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup._get_dynamics_pattern")
    def _get_dynamics_pattern(self, dynamics_type: str) -> str:
        """
        Get the pattern associated with a dynamics type.

        Args:
            dynamics_type: Type of dynamics (suma, decreciente, constante, indefinido)

        Returns:
            Pattern string for the dynamics type
        """
        patterns = {
            'suma': 'suma|total|agregado|consolidado',
            'decreciente': 'reducir|disminuir|decrementar|bajar',
            'constante': 'mantener|sostener|preservar|conservar',
            'indefinido': 'por definir|sin especificar|indefinido'
        }
        return patterns.get(dynamics_type, '')

class ReportingEngine:
    """Generate visualizations and reports"""

    def __init__(self, config: ConfigLoader, output_dir: Path) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.output_dir = output_dir
        self.output_dir.mkdir(parents=True, exist_ok=True)

    @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram")
    def generate_causal_diagram(self, graph: nx.DiGraph, policy_code: str) -> Path:
        """Generate causal diagram visualization"""
        dot = Dot(graph_type='digraph', rankdir='TB')
        dot.set_name(f'{policy_code}_causal_model')
        dot.set_node_defaults(
            shape='box',
            style='rounded,filled',
            fontname='Arial',
            fontsize='10'
        )
        dot.set_edge_defaults(
            fontsize='8',
            fontname='Arial'
        )

        # Add nodes with rigor coloring
        for node_id in graph.nodes():
            node_data = graph.nodes[node_id]

            # Determine color based on rigor status and audit flags
            rigor = node_data.get('rigor_status', 'sin_evaluar')
            audit_flags = node_data.get('audit_flags', [])
            financial = node_data.get('financial_allocation')

            if rigor == 'dÃ©bil' or not financial:
```

```
                color = 'lightcoral'  # Red
            elif audit_flags:
                color = 'lightyellow'  # Yellow
            else:
                color = 'lightgreen'  # Green

            # Create label
            node_type = node_data.get('type', 'programa')
            text = node_data.get('text', '')[:80]
            label = f"{node_id}\\n[{node_type.upper()}]\\n{text}..."

            entity = node_data.get('responsible_entity')
            if entity:
                label += f"\\nð\237\221¤ {entity[:30]}"

            if financial:
                label += f"\\nð\237\222° ${financial:,.0f}"

            dot_node = Node(
                node_id,
                label=label,
                fillcolor=color
            )
            dot.add_node(dot_node)

        # Add edges with causal logic
        for source, target in graph.edges():
            edge_data = graph.edges[source, target]
            keyword = edge_data.get('keyword', '')
            strength = edge_data.get('strength', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram", "default_strength", 0.5))

            # Determine edge style based on strength
            style = 'solid' if strength > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram", "solid_strength_threshold", 0.7) else 'dashed'

            dot_edge = Edge(
                source,
                target,
                label=keyword[:20],
                style=style
            )
            dot.add_edge(dot_edge)

        # Save files
        dot_path = self.output_dir / f"{policy_code}_causal_diagram.dot"
        png_path = self.output_dir / f"{policy_code}_causal_diagram.png"

        try:
            with open(dot_path, "w", encoding="utf-8") as f:
                f.write(dot.to_string())
            self.logger.info(f"Diagrama DOT guardado en: {dot_path}")

            # Try to render PNG
            try:
                dot.write_png(str(png_path))
```

```
                self.logger.info(f"Diagrama PNG renderizado en: {png_path}")
            except Exception as e:
                self.logger.warning(f"No se pudo renderizar PNG (Â¿Graphviz instalado?): {e}")
        except Exception as e:
            self.logger.error(f"Error guardando diagrama: {e}")

        return png_path

    @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_accountability_matrix")
    def generate_accountability_matrix(self, graph: nx.DiGraph,
                                       policy_code: str) -> Path:
        """Generate accountability matrix in Markdown"""
        md_path = self.output_dir / f"{policy_code}_accountability_matrix.md"

        # Group by impact goals
        impact_goals = [n for n in graph.nodes()
                        if graph.nodes[n].get(’type’) == ’impacto’]

        content = [f"# Matriz de Responsabilidades - {policy_code}\n"]
        content.append("*Generado automÃ¡ticamente por CDAF v2.0*\n")
        content.append("---\n\n")

        for impact in impact_goals:
            impact_data = graph.nodes[impact]
            content.append(f"## Meta de Impacto: {impact}\n")
            content.append(f"**DescripciÃ³n:** {impact_data.get(’text’, ’N/A’)}\n\n")

            # Find all predecessor chains
            predecessors = list(nx.ancestors(graph, impact))

            if predecessors:
                content.append("| Meta | Tipo | Entidad Responsable | Actividad Clave | Presupuesto |\n")
                content.append("|------|------|---------------------|-----------------|-------------|\n")

                for pred in predecessors:
                    pred_data = graph.nodes[pred]
                    meta_type = pred_data.get(’type’, ’N/A’)
                    entity = pred_data.get(’responsible_entity’, ’No asignado’)

                    ea = pred_data.get(’entity_activity’)
                    activity = ’N/A’
                    if ea and isinstance(ea, dict):
                        activity = ea.get(’activity’, ’N/A’)

                    budget = pred_data.get(’financial_allocation’)
                    budget_str = f"${budget:,.0f}" if budget else "Sin presupuesto"

                    content.append(f"| {pred} | {meta_type} | {entity} | {activity} | {budget_str} |\n")

                content.append("\n")
            else:
                content.append("*No se encontraron metas intermedias.*\n\n")

        content.append("\n---\n")
        content.append("### Leyenda\n")
        content.append("- **Meta de Impacto:** Resultado final esperado\n")
```

```
        content.append("- **Meta de Resultado:** Cambio intermedio observable\n")
        content.append("- **Meta de Producto:** Entrega tangible del programa\n")

        try:
            with open(md_path, 'w', encoding='utf-8') as f:
                f.write(''.join(content))
            self.logger.info(f"Matriz de responsabilidades guardada en: {md_path}")
        except Exception as e:
            self.logger.error(f"Error guardando matriz de responsabilidades: {e}")

        return md_path

    @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_confidence_report")
    def generate_confidence_report(self,
                                   nodes: dict[str, MetaNode],
                                   graph: nx.DiGraph,
                                   causal_chains: list[CausalLink],
                                   audit_results: dict[str, AuditResult],
                                   financial_auditor: FinancialAuditor,
                                   sequence_warnings: list[str],
                                   policy_code: str) -> Path:
        """Generate extraction confidence report"""
        json_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"

        # Calculate metrics
        total_metas = len(nodes)

        metas_with_ea = sum(1 for n in nodes.values() if n.entity_activity)
        metas_with_ea_pct = (metas_with_ea / total_metas * 100) if total_metas > 0 else 0

        enlaces_with_logic = sum(1 for link in causal_chains if link.get('logic'))
        total_edges = graph.number_of_edges()
        enlaces_with_logic_pct = (enlaces_with_logic / total_edges * 100) if total_edges > 0 else 0

        metas_passed_audit = sum(1 for r in audit_results.values() if r['passed'])
        metas_with_traceability_pct = (metas_passed_audit / total_metas * 100) if total_metas > 0 else 0

        metas_with_financial = sum(1 for n in nodes.values() if n.financial_allocation)
        metas_with_financial_pct = (metas_with_financial / total_metas * 100) if total_metas > 0 else 0

        # Node type distribution
        type_distribution = defaultdict(int)
        for node in nodes.values():
            type_distribution[node.type] += 1

        # Rigor distribution
        rigor_distribution = defaultdict(int)
        for node in nodes.values():
            rigor_distribution[node.rigor_status] += 1

        report = {
            "metadata": {
                "policy_code": policy_code,
                "framework_version": "2." + str(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_confidence_report", "framework_version", 0)),
                "total_nodes": total_metas,
```

```
                "total_edges": total_edges
            },
            "extraction_metrics": {
                "total_metas_identificadas": total_metas,
                "metas_con_EA_extraido": metas_with_ea,
                "metas_con_EA_extraido_pct": round(metas_with_ea_pct, 2),
                "enlaces_con_logica_causal": enlaces_with_logic,
                "enlaces_con_logica_causal_pct": round(enlaces_with_logic_pct, 2),
                "metas_con_trazabilidad_evidencia": metas_passed_audit,
                "metas_con_trazabilidad_evidencia_pct": round(metas_with_traceability_pct, 2),
                "metas_con_trazabilidad_financiera": metas_with_financial,
                "metas_con_trazabilidad_financiera_pct": round(metas_with_financial_pct, 2)
            },
            "financial_audit": {
                "tablas_financieras_parseadas_exitosamente": financial_auditor.successful_parses,
                "tablas_financieras_fallidas": financial_auditor.failed_parses,
                "asignaciones_presupuestarias_rastreadas": len(financial_auditor.financial_data)
            },
            "sequence_audit": {
                "alertas_secuencia_logica": len(sequence_warnings),
                "detalles": sequence_warnings
            },
            "type_distribution": dict(type_distribution),
            "rigor_distribution": dict(rigor_distribution),
            "audit_summary": {
                "total_audited": len(audit_results),
                "passed": sum(1 for r in audit_results.values() if r[’passed’]),
                "failed": sum(1 for r in audit_results.values() if not r[’passed’]),
                "total_warnings": sum(len(r[’warnings’]) for r in audit_results.values()),
                "total_errors": sum(len(r[’errors’]) for r in audit_results.values())
            },
            "quality_score": self._calculate_quality_score(
                metas_with_traceability_pct,
                metas_with_financial_pct,
                enlaces_with_logic_pct,
                metas_with_ea_pct
            )
        }

        try:
            with open(json_path, ’w’, encoding=’utf-8’) as f:
                json.dump(report, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Reporte de confianza guardado en: {json_path}")
        except Exception as e:
            self.logger.error(f"Error guardando reporte de confianza: {e}")

        return json_path

    @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score")
    def _calculate_quality_score(self, traceability: float, financial: float,
                                 logic: float, ea: float) -> float:
        """Calculate overall quality score (0-100)"""
        weights = {’traceability’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "traceability_weight", 0.35), ’financial’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "financial_weight", 0.25), ’logic’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "logic_weight", 0.25), ’ea’: ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "ea_weight", 0.15)}
```

```
        score = (traceability * weights['traceability'] +
                 financial * weights['financial'] +
                 logic * weights['logic'] +
                 ea * weights['ea'])
        return round(score, 2)

    @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_model_json")
    def generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
                                   policy_code: str) -> Path:
        """Generate structured JSON export of causal model"""
        json_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"

        # Prepare node data
        nodes_data = {}
        for node_id, node in nodes.items():
            node_dict = asdict(node)
            # Convert NamedTuple to dict
            if node.entity_activity:
                node_dict['entity_activity'] = node.entity_activity._asdict()
            nodes_data[node_id] = node_dict

        # Prepare edge data
        edges_data = []
        for source, target in graph.edges():
            edge_dict = {
                'source': source,
                'target': target,
                **graph.edges[source, target]
            }
            edges_data.append(edge_dict)

        model_data = {
            "policy_code": policy_code,
            "framework_version": "2." + str(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_model_json", "framework_version", 0)),
            "nodes": nodes_data,
            "edges": edges_data,
            "statistics": {
                "total_nodes": len(nodes_data),
                "total_edges": len(edges_data),
                "node_types": {
                    node_type: sum(1 for n in nodes.values() if n.type == node_type)
                    for node_type in ['programa', 'producto', 'resultado', 'impacto']
                }
            }
        }

        try:
            with open(json_path, 'w', encoding='utf-8') as f:
                json.dump(model_data, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Modelo causal JSON guardado en: {json_path}")
        except Exception as e:
            self.logger.error(f"Error guardando modelo causal: {e}")

        return json_path

```

```
class CDAFFramework:
    """Main orchestrator for the CDAF pipeline"""

    def __init__(self, config_path: Path, output_dir: Path, log_level: str = "INFO") -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.setLevel(getattr(logging, log_level.upper()))

        # Initialize components
        self.config = ConfigLoader(config_path)
        self.output_dir = output_dir

        # Initialize retry handler for external dependencies
        try:
            from retry_handler import DependencyType, get_retry_handler
            self.retry_handler = get_retry_handler()
            retry_enabled = True
        except ImportError:
            self.logger.warning("RetryHandler no disponible, funcionando sin retry logic")
            self.retry_handler = None
            retry_enabled = False

        # Load spaCy model with retry logic
        if retry_enabled and self.retry_handler:
            @self.retry_handler.with_retry(
                DependencyType.SPACY_MODEL,
                operation_name="load_spacy_model",
                exceptions=(OSError, IOError, ImportError)
            )
            def load_spacy_with_retry():
                try:
                    nlp = spacy.load("es_core_news_lg")
                    self.logger.info("Modelo spaCy cargado: es_core_news_lg")
                    return nlp
                except OSError:
                    self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando es_core_news_sm...")
                    nlp = spacy.load("es_core_news_sm")
                    return nlp
            try:
                self.nlp = load_spacy_with_retry()
            except OSError:
                self.logger.error("No se encontrÃ³ ningÃºn modelo de spaCy en espaÃ±ol. "
                                  "Ejecute: python -m spacy download es_core_news_lg")
                sys.exit(1)
        else:
            # Fallback to original logic without retry
            try:
                self.nlp = spacy.load("es_core_news_lg")
                self.logger.info("Modelo spaCy cargado: es_core_news_lg")
            except OSError:
                self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando es_core_news_sm...")
                try:
                    self.nlp = spacy.load("es_core_news_sm")
                except OSError:
                    self.logger.error("No se encontrÃ³ ningÃºn modelo de spaCy en espaÃ±ol. "
                                      "Ejecute: python -m spacy download es_core_news_lg")
                    sys.exit(1)
```

```

        # Initialize modules (pass retry_handler to PDF processor)
        self.pdf_processor = PDFProcessor(self.config, retry_handler=self.retry_handler if retry_enabled else None)
        self.causal_extractor = CausalExtractor(self.config, self.nlp)
        self.mechanism_extractor = MechanismPartExtractor(self.config, self.nlp)
        self.bayesian_mechanism = BayesianMechanismInference(self.config, self.nlp)
        self.financial_auditor = FinancialAuditor(self.config)
        self.op_auditor = OperationalizationAuditor(self.config)
        self.inference_setup = CausalInferenceSetup(self.config)
        self.reporting_engine = ReportingEngine(self.config, output_dir)

        # Initialize DNP validator if available
        self.dnp_validator = None
        if DNP_AVAILABLE:
            self.dnp_validator = ValidadorDNP(es_municipio_pdet=False)  # Can be configured
            self.logger.info("Validador DNP inicializado")

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework.process_document")
    def process_document(self, pdf_path: Path, policy_code: str) -> bool:
        """Main processing pipeline"""
        self.logger.info(f"Iniciando procesamiento de documento: {pdf_path}")

        try:
            # Step 1: Load and extract PDF
            if not self.pdf_processor.load_document(pdf_path):
                return False

            text = self.pdf_processor.extract_text()
            tables = self.pdf_processor.extract_tables()
            self.pdf_processor.extract_sections()

            # Step 2: Extract causal hierarchy
            self.logger.info("Extrayendo jerarquÃ-a causal...")
            graph = self.causal_extractor.extract_causal_hierarchy(text)
            nodes = self.causal_extractor.nodes

            # Step 3: Extract Entity-Activity pairs
            self.logger.info("Extrayendo tuplas Entidad-Actividad...")
            for node in nodes.values():
                if node.type == ’producto’:
                    ea = self.mechanism_extractor.extract_entity_activity(node.text)
                    if ea:
                        node.entity_activity = ea
                        graph.nodes[node.id][’entity_activity’] = ea._asdict()

            # Step 4: Financial traceability
            self.logger.info("Auditando trazabilidad financiera...")
            self.financial_auditor.trace_financial_allocation(tables, nodes, graph)

            # Step 4.5: Bayesian Mechanism Inference (AGUJA II)
            self.logger.info("Infiriendo mecanismos causales con modelo Bayesiano...")
            inferred_mechanisms = self.bayesian_mechanism.infer_mechanisms(nodes, text)

            # Step 5: Operationalization audit
            self.logger.info("Auditando operacionalizaciÃ³n...")
            audit_results = self.op_auditor.audit_evidence_traceability(nodes)
```

```
        sequence_warnings = self.op_auditor.audit_sequence_logic(graph)

        # Step 5.5: Bayesian Counterfactual Audit (AGUJA III)
        # Note: pdet_alignment should be calculated separately if needed via financiero_viabilidad_tablas
        # For now, using None as placeholder - can be enhanced by integrating PDETMunicipalPlanAnalyzer
        self.logger.info("Ejecutando auditorÃ-a contrafactual Bayesiana...")
        counterfactual_audit = self.op_auditor.bayesian_counterfactual_audit(nodes, graph, pdet_alignment=None)

        # Step 6: Causal inference setup
        self.logger.info("Preparando para inferencia causal...")
        self.inference_setup.classify_goal_dynamics(nodes)
        self.inference_setup.assign_probative_value(nodes)
        self.inference_setup.identify_failure_points(graph, text)

        # Step 7: DNP Standards Validation (if available)
        if self.dnp_validator:
            self.logger.info("Validando cumplimiento de estÃ¡ndares DNP...")
            self._validate_dnp_compliance(nodes, graph, policy_code)

        # Step 8: Generate reports
        self.logger.info("Generando reportes y visualizaciones...")
        self.reporting_engine.generate_causal_diagram(graph, policy_code)
        self.reporting_engine.generate_accountability_matrix(graph, policy_code)
        self.reporting_engine.generate_confidence_report(
            nodes, graph, self.causal_extractor.causal_chains,
            audit_results, self.financial_auditor, sequence_warnings, policy_code
        )
        self.reporting_engine.generate_causal_model_json(graph, nodes, policy_code)

        # Step 8: Generate Bayesian inference reports
        self.logger.info("Generando reportes de inferencia Bayesiana...")
        self._generate_bayesian_reports(
            inferred_mechanisms, counterfactual_audit, policy_code
        )

        # Step 9: Self-reflective learning from audit results (frontier paradigm)
        if self.config.validated_config and self.config.validated_config.self_reflection.enable_prior_learning:
            self.logger.info("Actualizando priors con retroalimentaciÃ³n del anÃ¡lisis...")
            feedback_data = self._extract_feedback_from_audit(
                inferred_mechanisms, counterfactual_audit, audit_results
            )
            self.config.update_priors_from_feedback(feedback_data)

            # HARMONIC FRONT 4: Check uncertainty reduction criterion
            if hasattr(self.bayesian_mechanism, ’_mean_mechanism_uncertainty’):
                uncertainty_check = self.config.check_uncertainty_reduction_criterion(
                    self.bayesian_mechanism._mean_mechanism_uncertainty
                )
                self.logger.info(
                    f"Uncertainty criterion check: {uncertainty_check[’status’]} "
                    f"({uncertainty_check[’iterations_tracked’]}/10 iterations, "
                    f"{uncertainty_check[’reduction_percent’]:.2f}% reduction)"
                )

        self.logger.info(f"â\234\205 Procesamiento completado exitosamente para {policy_code}")
        return True
```

```

        except CDAFException as e:
            # Structured error handling with custom exceptions
            self.logger.error(f"Error CDAF: {e.message}")
            self.logger.error(f"Detalles: {json.dumps(e.to_dict(), indent=2)}")
            if not e.recoverable:
                raise
            return False
        except Exception as e:
            # Wrap unexpected errors in CDAFProcessingError
            raise CDAFProcessingError(
                "Error crÃ-tico en el procesamiento",
                details={'error': str(e), 'type': type(e).__name__},
                stage="document_processing",
                recoverable=False
            ) from e

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._extract_feedback_from_audit")
    def _extract_feedback_from_audit(self, inferred_mechanisms: dict[str, dict[str, Any]],
                                     counterfactual_audit: dict[str, Any],
                                     audit_results: dict[str, AuditResult]) -> dict[str, Any]:
        """
        Extract feedback data from audit results for self-reflective prior updating

        This implements the frontier paradigm of learning from audit results
        to improve future inference accuracy.

        HARMONIC FRONT 4 ENHANCEMENT:
        - Reduces mechanism_type_priors for mechanisms with implementation_failure flags
        - Tracks necessity/sufficiency test failures
        - Penalizes "miracle" mechanisms that fail counterfactual tests
        """
        feedback = {}

        # Extract mechanism type frequencies from successful inferences
        mechanism_frequencies = defaultdict(float)
        failure_frequencies = defaultdict(float)  # NEW: Track failures
        total_mechanisms = 0
        total_failures = 0

        # Get causal implications from audit
        causal_implications = counterfactual_audit.get('causal_implications', {})

        for node_id, mechanism in inferred_mechanisms.items():
            mechanism_type_dist = mechanism.get('mechanism_type', {})
            # Weight by confidence (coherence score)
            confidence = mechanism.get('coherence_score', 0.5)

            # Check for implementation_failure flags in audit results
            node_implications = causal_implications.get(node_id, {})
            causal_effects = node_implications.get('causal_effects', {})
            has_implementation_failure = 'implementation_failure' in causal_effects

            # Check necessity/sufficiency test results
            necessity_test = mechanism.get('necessity_test', {})
            sufficiency_test = mechanism.get('sufficiency_test', {})
```

```
            failed_necessity = not necessity_test.get('is_necessary', True)
            failed_sufficiency = not sufficiency_test.get('is_sufficient', True)

            # If mechanism failed tests or has implementation_failure flag
            if has_implementation_failure or failed_necessity or failed_sufficiency:
                total_failures += 1
                # Track which mechanism types are associated with failures
                for mech_type, prob in mechanism_type_dist.items():
                    failure_frequencies[mech_type] += prob * confidence
            else:
                # Only count successes for positive reinforcement
                for mech_type, prob in mechanism_type_dist.items():
                    mechanism_frequencies[mech_type] += prob * confidence
                    total_mechanisms += confidence

        # Normalize frequencies
        if total_mechanisms > 0:
            mechanism_frequencies = {
                k: v / total_mechanisms
                for k, v in mechanism_frequencies.items()
            }
            feedback['mechanism_frequencies'] = dict(mechanism_frequencies)

        # NEW: Calculate penalty factors for failed mechanism types
        if total_failures > 0:
            failure_frequencies = {
                k: v / total_failures
                for k, v in failure_frequencies.items()
            }
            feedback['failure_frequencies'] = dict(failure_frequencies)

            # Calculate penalty: reduce priors for frequently failing types
            penalty_factors = {}
            for mech_type, failure_freq in failure_frequencies.items():
                # Higher failure frequency = stronger penalty (0.7 to 0.95) reduction)
                penalty_factors[mech_type] = 0.95 - (failure_freq * 0.25)
            feedback['penalty_factors'] = penalty_factors

        # Add audit quality metrics for future reference
        feedback['audit_quality'] = {
            'total_nodes_audited': len(audit_results),
            'passed_count': sum(1 for r in audit_results.values() if r['passed']),
            'success_rate': sum(1 for r in audit_results.values() if r['passed']) / max(len(audit_results), 1),
            'failure_count': total_failures,  # NEW
            'failure_rate': total_failures / max(len(inferred_mechanisms), 1)  # NEW
        }

        # Track necessity/sufficiency failures for iterative validation loop
        necessity_failures = sum(1 for m in inferred_mechanisms.values()
                                 if not m.get('necessity_test', {}).get('is_necessary', True))
        sufficiency_failures = sum(1 for m in inferred_mechanisms.values()
                                   if not m.get('sufficiency_test', {}).get('is_sufficient', True))

        feedback['test_failures'] = {
            'necessity_failures': necessity_failures,
            'sufficiency_failures': sufficiency_failures
```

```
        }

        return feedback

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._validate_dnp_compliance")
    def _validate_dnp_compliance(self, nodes: dict[str, MetaNode],
                                 graph: nx.DiGraph, policy_code: str) -> None:
        """
        Validate DNP compliance for all nodes/projects
        Generates DNP compliance report
        """
        if not self.dnp_validator:
            return

        # Build project list from nodes
        proyectos = []
        for node_id, node in nodes.items():
            # Extract sector from responsible entity or type
            sector = "general"
            if node.responsible_entity:
                entity_lower = node.responsible_entity.lower()
                if "educaci" in entity_lower or "edu" in entity_lower:
                    sector = "educacion"
                elif "salud" in entity_lower:
                    sector = "salud"
                elif "agua" in entity_lower or "acueducto" in entity_lower:
                    sector = "agua_potable_saneamiento"
                elif (
                        "via" in entity_lower or "vial" in entity_lower or "transporte" in entity_lower or "infraestructura" in entity_lower):
                    sector = "vias_transporte"
                elif "agr" in entity_lower or "rural" in entity_lower:
                    sector = "desarrollo_agropecuario"

            # Infer indicators from node type
            indicadores = []
            if node.type == "producto":
                # Map to MGA product indicators based on sector
                if sector == "educacion":
                    indicadores = ["EDU-020", "EDU-021"]
                elif sector == "salud":
                    indicadores = ["SAL-020", "SAL-021"]
                elif sector == "agua_potable_saneamiento":
                    indicadores = ["APS-020", "APS-021"]
            elif node.type == "resultado":
                # Map to MGA result indicators
                if sector == "educacion":
                    indicadores = ["EDU-001", "EDU-002"]
                elif sector == "salud":
                    indicadores = ["SAL-001", "SAL-002"]
                elif sector == "agua_potable_saneamiento":
                    indicadores = ["APS-001", "APS-002"]

            proyectos.append({
                "nombre": node_id,
                "sector": sector,
                "descripcion": node.text[:200] if node.text else "",
```

```
                "indicadores": indicadores,
            "presupuesto": node.financial_allocation or ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CDAFFramework._validate_dnp_compliance", "default_presupuesto", 0.0),
                "es_rural": "rural" in node.text.lower() if node.text else False,
                "poblacion_victimas": "v ctima" in node.text.lower() if node.text else False
            })

        # Validate each project
        dnp_results = []
        for proyecto in proyectos:
            resultado = self.dnp_validator.validar_proyecto_integral(
                sector=proyecto["sector"],
                descripcion=proyecto["descripcion"],
                indicadores_propuestos=proyecto["indicadores"],
                presupuesto=proyecto["presupuesto"],
                es_rural=proyecto["es_rural"],
                poblacion_victimas=proyecto["poblacion_victimas"]
            )
            dnp_results.append({
                "proyecto": proyecto["nombre"],
                "resultado": resultado
            })

        # Generate DNP compliance report
        self._generate_dnp_report(dnp_results, policy_code)

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_dnp_report")
    def _generate_dnp_report(self, dnp_results: list[dict], policy_code: str) -> None:
        """Generate comprehensive DNP compliance report"""
        report_path = self.output_dir / f"{policy_code}{DNP_REPORT_SUFFIX}"

        total_proyectos = len(dnp_results)
        if total_proyectos == 0:
            return

        # Calculate aggregate statistics
        proyectos_excelente = sum(1 for r in dnp_results
                                  if r["resultado"].nivel_cumplimiento.value == "excelente")
        proyectos_bueno = sum(1 for r in dnp_results
                              if r["resultado"].nivel_cumplimiento.value == "bueno")
        proyectos_aceptable = sum(1 for r in dnp_results
                                  if r["resultado"].nivel_cumplimiento.value == "aceptable")
        proyectos_insuficiente = sum(1 for r in dnp_results
                                     if r["resultado"].nivel_cumplimiento.value == "insuficiente")

        score_promedio = sum(r["resultado"].score_total for r in dnp_results) / total_proyectos

        # Build report
        lines = []
        lines.append("=" * 100)
        lines.append("REPORTE DE CUMPLIMIENTO DE ESTÃ\201NDARES DNP")
        lines.append(f"CÃ³digo de PolÃ-tica: {policy_code}")
        lines.append("=" * 100)
        lines.append("")

        lines.append("RESUMEN EJECUTIVO")
```

```
        lines.append("-" * 100)
        lines.append(f"Total de Proyectos/Metas Analizados: {total_proyectos}")
        lines.append(f"Score Promedio de Cumplimiento: {score_promedio:.1f}/100")
        lines.append("")
        lines.append("DistribuciÃ³n por Nivel de Cumplimiento:")
        lines.append(
            f"  â\200¢ Excelente (>90%):      {proyectos_excelente:3d} ({proyectos_excelente / total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  â\200¢ Bueno (75-90%):        {proyectos_bueno:3d} ({proyectos_bueno / total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  â\200¢ Aceptable (60-75%):    {proyectos_aceptable:3d} ({proyectos_aceptable / total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  â\200¢ Insuficiente (<60%):   {proyectos_insuficiente:3d} ({proyectos_insuficiente / total_proyectos * 100:5.1f}%)")
        lines.append("")

        # Detailed validation per project
        lines.append("VALIDACIÃ\223N DETALLADA POR PROYECTO/META")
        lines.append("=" * 100)

        for i, result_data in enumerate(dnp_results, 1):
            proyecto = result_data["proyecto"]
            resultado = result_data["resultado"]

            lines.append("")
            lines.append(f"{i}. {proyecto}")
            lines.append("-" * 100)
            lines.append(
                f"   Score: {resultado.score_total:.1f}/100 │ Nivel: {resultado.nivel_cumplimiento.value.upper()}")

            # Competencias
            comp_status = "â\234\223" if resultado.cumple_competencias else "â\234\227"
            lines.append(f"   Competencias Municipales: {comp_status}")
            if resultado.competencias_validadas:
                lines.append(f"     - Aplicables: {’, ’.join(resultado.competencias_validadas[:3])}")

            # MGA Indicators
            mga_status = "â\234\223" if resultado.cumple_mga else "â\234\227"
            lines.append(f"   Indicadores MGA: {mga_status}")
            if resultado.indicadores_mga_usados:
                lines.append(f"     - Usados: {’, ’.join(resultado.indicadores_mga_usados)}")
            if resultado.indicadores_mga_faltantes:
                lines.append(f"     - Recomendados: {’, ’.join(resultado.indicadores_mga_faltantes)}")

            # PDET (if applicable)
            if resultado.es_municipio_pdet:
                pdet_status = "â\234\223" if resultado.cumple_pdet else "â\234\227"
                lines.append(f"   Lineamientos PDET: {pdet_status}")
                if resultado.lineamientos_pdet_cumplidos:
                    lines.append(f"     - Cumplidos: {len(resultado.lineamientos_pdet_cumplidos)}")

            # Critical alerts
            if resultado.alertas_criticas:
                lines.append("   â\232  ALERTAS CRÃ\215TICAS:")
                for alerta in resultado.alertas_criticas:
                    lines.append(f"     - {alerta}")

```

```
            # Recommendations
            if resultado.recomendaciones:
                lines.append("   ð\237\223\213 RECOMENDACIONES:")
                for rec in resultado.recomendaciones[:3]:  # Top 3
                    lines.append(f"     - {rec}")

        lines.append("")
        lines.append("=" * 100)
        lines.append("NORMATIVA DE REFERENCIA")
        lines.append("-" * 100)
        lines.append("â\200¢ Competencias Municipales: Ley 136/1994, Ley 715/2001, Ley 1551/2012")
        lines.append("â\200¢ Indicadores MGA: DNP - MetodologÃa General Ajustada")
        lines.append("â\200¢ PDET: Decreto 893/2017, Acuerdo Final de Paz")
        lines.append("=" * 100)

        # Write report
        try:
            with open(report_path, 'w', encoding='utf-8') as f:
                f.write('\n'.join(lines))
            self.logger.info(f"Reporte de cumplimiento DNP guardado en: {report_path}")
        except Exception as e:
            self.logger.error(f"Error guardando reporte DNP: {e}")

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._audit_causal_coherence")
    def _audit_causal_coherence(self, graph: nx.DiGraph, nodes: dict[str, MetaNode]) -> dict[str, Any]:
        """
        Audit causal coherence of the extracted model.

        Args:
            graph: Causal graph
            nodes: Dictionary of nodes

        Returns:
            Dictionary with coherence audit results
        """
        audit = {
            'total_nodes': len(nodes),
            'total_edges': graph.number_of_edges(),
            'disconnected_nodes': [],
            'cycles': [],
            'coherence_score': 0.0
        }

        # Check for disconnected nodes
        for node_id in nodes:
            if graph.has_node(node_id) and graph.degree(node_id) == 0:
                audit['disconnected_nodes'].append(node_id)

        # Check for cycles (should not exist in causal DAG)
        try:
            cycles = list(nx.simple_cycles(graph))
            audit['cycles'] = cycles
        except:
            pass

        # Calculate coherence score
```

```
        connected_ratio = 1.0 - (len(audit['disconnected_nodes']) / max(len(nodes), 1))
        acyclic_score = 1.0 if len(audit['cycles']) == 0 else 0.5
        audit['coherence_score'] = (connected_ratio + acyclic_score) / 2.0

        return audit

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_causal_model_json")
    def _generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
                                    policy_code: str) -> None:
        """
        Generate JSON representation of causal model.

        Args:
            graph: Causal graph
            nodes: Dictionary of nodes
            policy_code: Policy code for filename
        """
        model = {
            'policy_code': policy_code,
            'nodes': [],
            'edges': []
        }

        # Add nodes
        for node_id, node in nodes.items():
            model['nodes'].append({
                'id': node_id,
                'text': node.text,
                'type': node.type,
                'baseline': str(node.baseline) if node.baseline else None,
                'target': str(node.target) if node.target else None
            })

        # Add edges
        for source, target in graph.edges():
            edge_data = graph.get_edge_data(source, target)
            model['edges'].append({
                'source': source,
                'target': target,
                'logic': edge_data.get('logic', 'unknown'),
                'strength': edge_data.get('strength', 0.5)
            })

        # Write to file
        output_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"
        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(model, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Causal model JSON saved to: {output_path}")
        except Exception as e:
            self.logger.error(f"Error saving causal model JSON: {e}")

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_dnp_compliance_report")
    def _generate_dnp_compliance_report(self, nodes: dict[str, MetaNode],
                                        policy_code: str) -> dict[str, Any]:
        """
```

```
        Generate DNP compliance report.

        Args:
            nodes: Dictionary of nodes
            policy_code: Policy code

        Returns:
            Compliance report dictionary
        """
        report = {
            'policy_code': policy_code,
            'total_products': 0,
            'compliant_products': 0,
            'compliance_rate': 0.0,
            'gaps': []
        }

        # Check products for DNP compliance
        for node_id, node in nodes.items():
            if node.type == 'producto':
                report['total_products'] += 1

                # Check required fields
                has_baseline = node.baseline is not None
                has_target = node.target is not None
                has_indicator = len(node.text) > 10  # Simple check

                is_compliant = has_baseline and has_target and has_indicator

                if is_compliant:
                    report['compliant_products'] += 1
                else:
                    gaps = []
                    if not has_baseline:
                        gaps.append('missing_baseline')
                    if not has_target:
                        gaps.append('missing_target')
                    if not has_indicator:
                        gaps.append('missing_indicator')

                    report['gaps'].append({
                        'node_id': node_id,
                        'issues': gaps
                    })

        if report['total_products'] > 0:
            report['compliance_rate'] = report['compliant_products'] / report['total_products']

        return report

    @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_extraction_report")
    def _generate_extraction_report(self, nodes: dict[str, MetaNode],
                                    graph: nx.DiGraph,
                                    policy_code: str) -> None:
        """
        Generate extraction confidence report.
```

```

        Args:
            nodes: Dictionary of nodes
            graph: Causal graph
            policy_code: Policy code
        """
        report = {
            'policy_code': policy_code,
            'extraction_summary': {
                'total_nodes': len(nodes),
                'total_edges': graph.number_of_edges(),
                'nodes_by_type': {}
            },
            'node_confidence': []
        }

        # Count nodes by type
        for node in nodes.values():
            node_type = node.type
            report['extraction_summary']['nodes_by_type'][node_type] = \
                report['extraction_summary']['nodes_by_type'].get(node_type, 0) + 1

        # Add confidence scores
        for node_id, node in nodes.items():
            confidence = 0.8 # Refactored
            if hasattr(node, 'rigor_status'):
                if node.rigor_status == 'fuerte':
                    confidence = 0.9 # Refactored
                elif node.rigor_status == 'dÃ©bil':
                    confidence = 0.6 # Refactored

            report['node_confidence'].append({
                'node_id': node_id,
                'confidence': confidence
            })

        # Write report
        output_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"
        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(report, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Extraction report saved to: {output_path}")
        except Exception as e:
            self.logger.error(f"Error saving extraction report: {e}")

# ============================================================================
# AGUJA I: PRIOR ADAPTATIVO (EVIDENCIA â\206\222 BAYES)
# ============================================================================

class BayesFactorTable:
    """Tabla fija de Bayes Factors por tipo de test evidencial (Beach & Pedersen 2019)"""
    FACTORS = {
        'straw': (1.0, 1.5),      # STRAW_IN_WIND: Weak evidence
        'hoop': (3.0, 5.0),       # HOOP TEST: Necessary but not sufficient
        'smoking': (10.0, 30.0),  # SMOKING GUN: Sufficient but not necessary
        'doubly': (50.0, 100.0)   # DOUBLY DECISIVE: Necessary AND sufficient
```

```
    }

    @classmethod
    def get_bayes_factor(cls, test_type: str) -> float:
        """Obtiene BF medio para tipo de test"""
        if test_type not in cls.FACTORS:
            return 1.5  # Default straw-in-wind
        min_bf, max_bf = cls.FACTORS[test_type]
        return (min_bf + max_bf) / 2.0

    @classmethod
    def get_version(cls) -> str:
        """Version de tabla BF para trazabilidad"""
        return "Beach2019_v1.0)"

class AdaptivePriorCalculator:
    """
    AGUJA I - Prior Adaptativo con Bayes Factor y calibraciÃ³n

    PROMPT I-1: PonderaciÃ³n evidencial con BF y calibraciÃ³n
    Mapea test_typeâ\206\222BayesFactor, calcula likelihood adaptativo combinando
    dominios {semantic, temporal, financial, structural} con pesos normalizados.

    PROMPT I-2: Sensibilidad, OOD y ablation evidencial
    Perturba cada componente Â±10% y reporta â\210\202p/â\210\202component top-3.

    PROMPT I-3: Trazabilidad y reproducibilidad
    Con semilla fija, guarda bf_table_version, weights_version, snippets.

    QUALITY CRITERIA:
    - BrierScore â\211¤ 0.20) en validaciÃ³n sintÃ©tica
    - ACE â\210\210 [â\210\2220.02), 0.02)] (Average Calibration Error)
    - Cobertura CI95% â\210\210 [92%, 98%]
    - Monotonicidad: â\206\221 seÃ±ales â\206\222 Â¬â\206\223 p_mechanism
    """

    def __init__(self, calibration_params: dict[str, float] | None = None) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.bf_table = BayesFactorTable()

        # Calibration params: logitâ\201»Â¹(Î± + Î²Â•score)
        self.calibration = calibration_params or {
            ’alpha’: -2.0,  # Intercept
            ’beta’: 4.0     # Slope
        }

        # Domain weights (normalized)
        self.default_domain_weights = {
            ’semantic’: 0.35,
            ’temporal’: 0.25,
            ’financial’: 0.25,
            ’structural’: 0.15
        }

    def calculate_likelihood_adaptativo(
        self,
```

```
        evidence_dict: dict[str, Any],
        test_type: str = ’hoop’
    ) -> dict[str, Any]:
        """
        PROMPT I-1: Calcula likelihood adaptativo con BF y dominios

        Args:
            evidence_dict: Evidencia por caso {semantic, temporal, financial, structural}
            test_type: Tipo de test evidencial (straw, hoop, smoking, doubly)

        Returns:
            Dict con p_mechanism, BF_used, domain_weights, triangulation_bonus, etc.
        """
        # 1. Obtener Bayes Factor para test_type
        bf_used = self.bf_table.get_bayes_factor(test_type)

        # 2. Extraer scores por dominio
        domain_scores = {
            ’semantic’: evidence_dict.get(’semantic’, {}).get(’score’, 0.0),
            ’temporal’: evidence_dict.get(’temporal’, {}).get(’score’, 0.0),
            ’financial’: evidence_dict.get(’financial’, {}).get(’score’, 0.0),
            ’structural’: evidence_dict.get(’structural’, {}).get(’score’, 0.0)
        }

        # 3. Ajustar pesos si falta dominio (baja peso a 0, reparte)
        adjusted_weights = self._adjust_domain_weights(domain_scores)

        # 4. Calcular score combinado normalizado
        combined_score = sum(
            domain_scores[domain] * adjusted_weights[domain]
            for domain in domain_scores
        )

        # 5. Aplicar multiplicador BF normalizado
        all_bfs = [np.mean(bf_range) for bf_range in self.bf_table.FACTORS.values()]
        mean_bf = np.mean(all_bfs)
        bf_multiplier = bf_used / mean_bf
        adapted_score = combined_score * bf_multiplier

        # 6. Bonus de triangulaciÃ³n si â\211¥3 dominios activos
        active_domains = sum(1 for s in domain_scores.values() if s > 0.1)
        triangulation_bonus = 0.05 if active_domains >= 3 else 0.0

        final_score = min(1.0, adapted_score + triangulation_bonus)

        # 7. Transformar a probabilidad con logit inverso: p = 1/(1+exp(-(Î±+Î²Â•score)))
        alpha = self.calibration[’alpha’]
        beta = self.calibration[’beta’]
        logit_value = alpha + beta * final_score
        p_mechanism = 1.0 / (1.0 + np.exp(-logit_value))

        # 8. Clip [1e-6, 1-1e-6]
        p_mechanism = np.clip(p_mechanism, 1e-6, 1 - 1e-6)

        return {
            ’p_mechanism’: float(p_mechanism),
```

```
            'BF_used': bf_used,
            'domain_weights': adjusted_weights,
            'triangulation_bonus': triangulation_bonus,
            'calibration_params': self.calibration,
            'test_type': test_type,
            'combined_score': combined_score,
            'active_domains': active_domains
        }

    @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator._adjust_domain_weights")
    def _adjust_domain_weights(self, domain_scores: dict[str, float]) -> dict[str, float]:
        """Ajusta pesos si falta dominio: baja a 0 y reparte"""
        adjusted = self.default_domain_weights.copy()

        # Identificar dominios faltantes (score â\211¤ 0)
        missing_domains = [d for d, s in domain_scores.items() if s <= 0]

        if missing_domains:
            # Bajar peso a 0 para dominios faltantes
            total_missing_weight = sum(adjusted[d] for d in missing_domains)
            for d in missing_domains:
                adjusted[d] = 0.0

            # Repartir peso entre dominios activos
            active_domains = [d for d in adjusted if adjusted[d] > 0]
            if active_domains:
                bonus_per_domain = total_missing_weight / len(active_domains)
                for d in active_domains:
                    adjusted[d] += bonus_per_domain

        # Renormalizar para asegurar suma = 1.0
        total = sum(adjusted.values())
        if total > 0:
            adjusted = {k: v / total for k, v in adjusted.items()}

        return adjusted

    def sensitivity_analysis(
        self,
        evidence_dict: dict[str, Any],
        test_type: str = 'hoop',
        perturbation: float = 0.10
    ) -> dict[str, Any]:
        """
        PROMPT I-2: Sensibilidad, OOD y ablation evidencial

        Perturba cada componente Â±10% y reporta â\210\202p/â\210\202component top-3.
        Ejecuta ablaciones: sÃ³lo textual, sÃ³lo financiero, sÃ³lo estructural.

        CRITERIA:
        - |delta_p_sensitivity|_max â\211¤ 0.15)
        - sign_concordance â\211¥ 2/3
        - OOD_drop â\211¤ 0.10)
        """
        # Baseline
        baseline_result = self.calculate_likelihood_adaptativo(evidence_dict, test_type)
```

```
        baseline_p = baseline_result[’p_mechanism’]

        # 1. Sensibilidad por componente
        sensitivity_map = {}
        for domain in [’semantic’, ’temporal’, ’financial’, ’structural’]:
            if domain in evidence_dict and isinstance(evidence_dict[domain], dict) and ’score’ in evidence_dict[domain]:
                # Perturbar +10%
                perturbed_evidence = self._perturb_evidence(evidence_dict, domain, perturbation)
                perturbed_result = self.calculate_likelihood_adaptativo(perturbed_evidence, test_type)
                delta_p = perturbed_result[’p_mechanism’] - baseline_p

                sensitivity_map[domain] = {
                    ’delta_p’: delta_p,
                    ’relative_change’: delta_p / max(baseline_p, 1e-6)
                }

        # Top-3 por magnitud
        top_3 = sorted(
            sensitivity_map.items(),
            key=lambda x: abs(x[1][’delta_p’]),
            reverse=True
        )[:3]

        # 2. Ablaciones: sÃ³lo un dominio
        ablation_results = {}
        for domain in [’semantic’, ’financial’, ’structural’]:
            ablated_evidence = {
                domain: evidence_dict.get(domain, {’score’: 0.0})
            }
            if ablated_evidence[domain].get(’score’, 0) > 0:
                abl_result = self.calculate_likelihood_adaptativo(ablated_evidence, test_type)
                ablation_results[f’only_{domain}’] = {
                    ’p_mechanism’: abl_result[’p_mechanism’],
                    ’sign_match’: (abl_result[’p_mechanism’] > 0.5) == (baseline_p > 0.5)
                }

        # Sign concordance
        sign_concordance = sum(
            1 for r in ablation_results.values() if r[’sign_match’]
        ) / max(len(ablation_results), 1)

        # 3. OOD con ruido
        ood_evidence = self._add_ood_noise(evidence_dict)
        ood_result = self.calculate_likelihood_adaptativo(ood_evidence, test_type)
        ood_drop = abs(baseline_p - ood_result[’p_mechanism’])

        # 4. EvaluaciÃ³n de criterios
        max_sensitivity = max((abs(item[1][’delta_p’]) for item in top_3), default=0.0)
        criteria_met = {
            ’max_sensitivity_ok’: max_sensitivity <= 0.15,
            ’sign_concordance_ok’: sign_concordance >= 2/3,
            ’ood_drop_ok’: ood_drop <= 0.10
        }

        # Determinar si caso es frÃ¡gil
        is_fragile = not all(criteria_met.values())
```

```

        return {
            'influence_top3': [(domain, data['delta_p']) for domain, data in top_3],
            'delta_p_sensitivity': max_sensitivity,
            'sign_concordance': sign_concordance,
            'OOD_drop': ood_drop,
            'ablation_results': ablation_results,
            'criteria_met': criteria_met,
            'is_fragile': is_fragile,
            'recommendation': 'downgrade' if is_fragile else 'accept'
        }

    def _perturb_evidence(
        self,
        evidence_dict: dict[str, Any],
        domain: str,
        perturbation: float
    ) -> dict[str, Any]:
        """Perturba un dominio especÃfico"""
        import copy
        perturbed = copy.deepcopy(evidence_dict)
        if domain in perturbed and isinstance(perturbed[domain], dict) and 'score' in perturbed[domain]:
            perturbed[domain]['score'] *= (1.0 + perturbation)
            perturbed[domain]['score'] = min(1.0, perturbed[domain]['score'])
        return perturbed

    @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator._add_ood_noise")
    def _add_ood_noise(self, evidence_dict: dict[str, Any]) -> dict[str, Any]:
        """Genera set OOD con ruido semÃ¡ntico y tablas malformadas"""
        import copy
        ood = copy.deepcopy(evidence_dict)

        # Agregar ruido gaussiano a todos los scores
        for domain in ood:
            if isinstance(ood[domain], dict) and 'score' in ood[domain]:
                noise = np.random.normal(0, 0.05)  # 5% noise
                ood[domain]['score'] = np.clip(ood[domain]['score'] + noise, 0.0, 1.0)

        return ood

    def generate_traceability_record(
        self,
        evidence_dict: dict[str, Any],
        test_type: str,
        result: dict[str, Any],
        seed: int = 42
    ) -> dict[str, Any]:
        """
        PROMPT I-3: Trazabilidad y reproducibilidad

        Con semilla fija, guarda bf_table_version, weights_version,
        snippets textuales con offsets, campos financieros usados.

        METRICS:
        - Re-ejecuciÃ³n con misma semilla produce hash_result idÃ©ntico
        - trace_completeness â\211¥ 0.95)
```

```
        """
        # Fijar semilla para reproducibilidad
        np.random.seed(seed)

        # Construir evidence trace
        evidence_trace = []
        for domain, data in evidence_dict.items():
            if isinstance(data, dict) and ’score’ in data:
                trace_item = {
                    ’source’: domain,
                    ’line_span’: data.get(’line_span’, ’unknown’),
                    ’transform_before’: data.get(’raw_value’, None),
                    ’transform_after’: data[’score’],
                    ’snippet’: data.get(’snippet’, ’’)[:100]  # Primeros 100 chars
                }
                evidence_trace.append(trace_item)

        # Config hash
        config_str = json.dumps({
            ’bf_table_version’: self.bf_table.get_version(),
            ’calibration_params’: self.calibration,
            ’domain_weights’: self.default_domain_weights,
            ’test_type’: test_type,
            ’seed’: seed
        }, sort_keys=True)

        config_hash = hashlib.sha256(config_str.encode()).hexdigest()[:16]

        # Result hash
        result_str = json.dumps(result, sort_keys=True)
        result_hash = hashlib.sha256(result_str.encode()).hexdigest()[:16]

        # Trace completeness
        factors_in_trace = len(evidence_trace)
        total_factors = len([d for d in evidence_dict if isinstance(evidence_dict.get(d), dict)])
        trace_completeness = factors_in_trace / max(total_factors, 1)

        return {
            ’evidence_trace’: evidence_trace,
            ’hash_config’: config_hash,
            ’hash_result’: result_hash,
            ’seed’: seed,
            ’bf_table_version’: self.bf_table.get_version(),
            ’weights_version’: ’default_v1.0’,
            ’trace_completeness’: trace_completeness,
            ’reproducibility_guaranteed’: trace_completeness >= 0.95
        }

    @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator.validate_quality_criteria")
    def validate_quality_criteria(self, validation_samples: list[dict[str, Any]]) -> dict[str, Any]:
        """
        Valida criterios de calidad en conjunto de validaciÃ³n sintÃ©tica

        QUALITY CRITERIA:
        - BrierScore â\211¤ 0.20)
        - ACE â\210\210 [â\210\2220.02), 0.02)]
```

```
        - Cobertura CI95% â\210\210 [92%, 98%]
        - Monotonicidad verificada
        """
        predictions = []
        actuals = []

        for sample in validation_samples:
            evidence = sample.get('evidence', {})
            actual_label = sample.get('actual_label', 0.5)
            test_type = sample.get('test_type', 'hoop')

            result = self.calculate_likelihood_adaptativo(evidence, test_type)
            predictions.append(result['p_mechanism'])
            actuals.append(actual_label)

        predictions = np.array(predictions)
        actuals = np.array(actuals)

        # 1. Brier Score
        brier_score = np.mean((predictions - actuals) ** 2)
        brier_ok = brier_score <= 0.20

        # 2. ACE (Average Calibration Error)
        # Dividir en bins
        n_bins = 10
        bin_boundaries = np.linspace(0, 1, n_bins + 1)
        ace = 0.0 # Refactored

        for i in range(n_bins):
            bin_mask = (predictions >= bin_boundaries[i]) & (predictions < bin_boundaries[i + 1])
            if bin_mask.sum() > 0:
                bin_accuracy = actuals[bin_mask].mean()
                bin_confidence = predictions[bin_mask].mean()
                ace += abs(bin_accuracy - bin_confidence) / n_bins

        ace_ok = -0.02 <= ace <= 0.02

        # 3. Cobertura CI95%
        # Simular con bootstrap
        n_bootstrap = 100
        coverage_count = 0

        for _ in range(n_bootstrap):
            idx = np.random.choice(len(predictions), size=len(predictions), replace=True)
            boot_preds = predictions[idx]
            boot_actuals = actuals[idx]

            # Calcular CI95%
            ci_low = np.percentile(boot_preds, 2.5)
            ci_high = np.percentile(boot_preds, 97.5)

            # Verificar si mean actual estÃ¡ dentro
            actual_mean = boot_actuals.mean()
            if ci_low <= actual_mean <= ci_high:
                coverage_count += 1

```

```
        coverage = coverage_count / n_bootstrap
        coverage_ok = 0.92 <= coverage <= 0.98

        # 4. Monotonicidad: verificar que â\206\221 seÃ±ales â\206\222 Â¬â\206\223 p_mechanism
        monotonicity_violations = 0

        for i in range(len(validation_samples) - 1):
            current_total = sum(
                validation_samples[i][’evidence’].get(d, {}).get(’score’, 0)
                for d in [’semantic’, ’temporal’, ’financial’, ’structural’]
            )
            next_total = sum(
                validation_samples[i + 1][’evidence’].get(d, {}).get(’score’, 0)
                for d in [’semantic’, ’temporal’, ’financial’, ’structural’]
            )

            if next_total > current_total and predictions[i + 1] < predictions[i]:
                monotonicity_violations += 1

        monotonicity_ok = monotonicity_violations == 0

        return {
            ’brier_score’: float(brier_score),
            ’brier_ok’: brier_ok,
            ’ace’: float(ace),
            ’ace_ok’: ace_ok,
            ’ci95_coverage’: float(coverage),
            ’coverage_ok’: coverage_ok,
            ’monotonicity_violations’: monotonicity_violations,
            ’monotonicity_ok’: monotonicity_ok,
            ’all_criteria_met’: brier_ok and ace_ok and coverage_ok and monotonicity_ok,
            ’quality_grade’: ’EXCELLENT’ if (brier_ok and ace_ok and coverage_ok and monotonicity_ok) else ’NEEDS_IMPROVEMENT’
        }

# ============================================================================
# AGUJA II: MODELO GENERATIVO JERÃ\201RQUICO
# ============================================================================

class HierarchicalGenerativeModel:
    """
    AGUJA II - Modelo Generativo JerÃ¡rquico con inferencia MCMC

    PROMPT II-1: Inferencia jerÃ¡rquica con incertidumbre
    Estima posterior(mechanism_type, activity_sequence | obs) con MCMC.

    PROMPT II-2: Posterior Predictive Checks + Ablation
    Genera datos simulados desde posterior y compara con observados.

    PROMPT II-3: Independencias y parsimonia
    Verifica d-separaciones y calcula Î\224WAIC.

    QUALITY CRITERIA:
    - R-hat â\211¤ 1.10
    - ESS â\211¥ 200
    - entropy/entropy_max < 0.7) para certeza
    - ppd_p_value â\210\210 [0.1), 0.9)]
```

```
    - Î\224WAIC â\211¤ â\210\2222 para preferir jerÃ¡rquico
    """

    def __init__(self, mechanism_priors: dict[str, float] | None = None) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)

        # Priors dÃ©biles para mechanism_type si no se proveen
        self.mechanism_priors = mechanism_priors or {
            ’administrativo’: 0.30,
            ’tecnico’: 0.25,
            ’financiero’: 0.20,
            ’politico’: 0.15,
            ’mixto’: 0.10
        }

        # Validar que suman ˜1.0
        prior_sum = sum(self.mechanism_priors.values())
        if abs(prior_sum - 1.0) > 0.01:
            self.logger.warning(f"Mechanism priors sum to {prior_sum:.3f}, normalizing...")
            self.mechanism_priors = {
                k: v / prior_sum for k, v in self.mechanism_priors.items()
            }

    def infer_mechanism_posterior(
        self,
        observations: dict[str, Any],
        n_iter: int = 500,
        burn_in: int = 100,
        n_chains: int = 2
    ) -> dict[str, Any]:
        """
        PROMPT II-1: Inferencia jerÃ¡rquica con MCMC

        Estima posterior(mechanism_type, activity_sequence | obs) usando MCMC.

        Args:
            observations: Dict con {verbos, co_ocurrencias, coherence, structural_signals}
            n_iter: Iteraciones MCMC (â\211¥500)
            burn_in: Burn-in iterations (â\211¥100)
            n_chains: NÃºmero de cadenas para R-hat (â\211¥2)

        Returns:
            Dict con type_posterior, sequence_mode, coherence_score, entropy, CI95, R-hat, ESS
        """
        self.logger.info(f"Starting MCMC inference: {n_iter} iter, {burn_in} burn-in, {n_chains} chains")

        # Validar observaciones mÃnimas
        if not observations or ’coherence’ not in observations:
            self.logger.warning("Missing observations, using weak priors")
            observations = observations or {}
            observations.setdefault(’coherence’, 0.5)

        # Ejecutar mÃºltiples cadenas para diagnÃ³stico
        chains = []
        for chain_idx in range(n_chains):
            chain_samples = self._run_mcmc_chain(
```

```
                observations, n_iter, burn_in, seed=42 + chain_idx
            )
            chains.append(chain_samples)
            self.logger.debug(f"Chain {chain_idx + 1}/{n_chains} completed: {len(chain_samples)} samples")

        # Agregar samples de todas las cadenas
        all_samples = []
        for chain in chains:
            all_samples.extend(chain)

        # 1. Type posterior (frecuencias de mechanism_type)
        type_counts = dict.fromkeys(self.mechanism_priors.keys(), 0)
        for sample in all_samples:
            mtype = sample.get('mechanism_type', 'mixto')
            if mtype in type_counts:
                type_counts[mtype] += 1

        total_samples = len(all_samples)
        type_posterior = {
            mtype: count / max(total_samples, 1)
            for mtype, count in type_counts.items()
        }

        # 2. Sequence mode (secuencia mÃ¡s frecuente)
        sequence_mode = self._get_mode_sequence(all_samples)

        # 3. Coherence score (estadÃsticas)
        coherence_scores = [s.get('coherence', 0.5) for s in all_samples]
        coherence_mean = float(np.mean(coherence_scores))
        coherence_std = float(np.std(coherence_scores))

        # 4. Entropy del posterior
        posterior_probs = list(type_posterior.values())
        entropy_posterior = -sum(p * np.log(p + 1e-10) for p in posterior_probs if p > 0)
        max_entropy = np.log(len(self.mechanism_priors))
        normalized_entropy = entropy_posterior / max_entropy if max_entropy > 0 else 0.0

        # 5. CI95 para coherence
        ci95_low = float(np.percentile(coherence_scores, 2.5))
        ci95_high = float(np.percentile(coherence_scores, 97.5))

        # 6. R-hat aproximado (between-chain variance / within-chain variance)
        r_hat = self._calculate_r_hat(chains)

        # 7. ESS (Effective Sample Size)
        ess = self._calculate_ess(all_samples)

        # 8. Verificar criterios de calidad
        is_uncertain = normalized_entropy > 0.7
        criteria_met = {
            'r_hat_ok': r_hat <= 1.10,
            'ess_ok': ess >= 200,
            'entropy_ok': not is_uncertain
        }

        # Warning si alta incertidumbre
```

```
        warning = None
        if is_uncertain:
            warning = f"HIGH_UNCERTAINTY: entropy/entropy_max = {normalized_entropy:.3f} > 0.7)"
            self.logger.warning(warning)

        return {
            'type_posterior': type_posterior,
            'sequence_mode': sequence_mode,
            'coherence_score': coherence_mean,
            'coherence_std': coherence_std,
            'entropy_posterior': float(entropy_posterior),
            'normalized_entropy': float(normalized_entropy),
            'CI95': (ci95_low, ci95_high),
            'CI95_width': ci95_high - ci95_low,
            'R_hat': float(r_hat),
            'ESS': float(ess),
            'n_samples': total_samples,
            'is_uncertain': is_uncertain,
            'criteria_met': criteria_met,
            'warning': warning
        }

    def _run_mcmc_chain(
        self,
        observations: dict[str, Any],
        n_iter: int,
        burn_in: int,
        seed: int
    ) -> list[dict[str, Any]]:
        """Ejecuta una cadena MCMC con Metropolis-Hastings"""
        np.random.seed(seed)
        samples = []

        # Estado inicial: sample desde prior
        current_type = np.random.choice(
            list(self.mechanism_priors.keys()),
            p=list(self.mechanism_priors.values())
        )
        current_coherence = observations.get('coherence', 0.5)

        for i in range(n_iter):
            # Proponer nuevo mechanism_type
            proposed_type = np.random.choice(list(self.mechanism_priors.keys()))

            # Calcular likelihood ratio
            current_likelihood = self._calculate_likelihood(current_type, observations)
            proposed_likelihood = self._calculate_likelihood(proposed_type, observations)

            # Prior ratio
            prior_ratio = self.mechanism_priors[proposed_type] / max(self.mechanism_priors[current_type], 1e-10)

            # Acceptance probability (Metropolis-Hastings)
            likelihood_ratio = proposed_likelihood / max(current_likelihood, 1e-10)
            acceptance_prob = min(1.0, likelihood_ratio * prior_ratio)

            # Accept/reject
```

```
            if np.random.random() < acceptance_prob:
                current_type = proposed_type

            # Simular coherence con ruido
            simulated_coherence = current_coherence + np.random.normal(0, 0.05)
            simulated_coherence = np.clip(simulated_coherence, 0.0, 1.0)

            # Almacenar sample (despuÃ©s de burn-in)
            if i >= burn_in:
                sample = {
                    'mechanism_type': current_type,
                    'coherence': float(simulated_coherence),
                    'iteration': i - burn_in,
                    'chain_seed': seed
                }
                samples.append(sample)

        return samples

    def _calculate_likelihood(
        self,
        mechanism_type: str,
        observations: dict[str, Any]
    ) -> float:
        """Calcula likelihood de observations dado mechanism_type"""
        # Likelihood basado en coherence y structural signals
        coherence = observations.get('coherence', 0.5)
        structural_signals = observations.get('structural_signals', {})

        # Base likelihood desde prior
        prior = self.mechanism_priors.get(mechanism_type, 0.1)

        # Ajuste por coherence (mayor coherence â\206\222 mayor likelihood)
        coherence_factor = 1.0 + coherence

        # Ajuste por seÃ±ales estructurales especÃficas del tipo
        structural_match = 0.0 # Refactored
        if mechanism_type == 'administrativo' and structural_signals.get('admin_keywords', 0) > 0:
            structural_match = 0.2 # Refactored
        elif mechanism_type == 'financiero' and structural_signals.get('budget_data', 0) > 0:
            structural_match = 0.3 # Refactored
        elif mechanism_type == 'tecnico' and structural_signals.get('technical_terms', 0) > 0:
            structural_match = 0.25 # Refactored

        likelihood = prior * coherence_factor * (1.0 + structural_match)
        return likelihood

    @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._get_mode_sequence")
    def _get_mode_sequence(self, samples: list[dict[str, Any]]) -> str:
        """Obtiene secuencia modal (tipo mÃ¡s frecuente)"""
        type_counts = {}
        for s in samples:
            mtype = s.get('mechanism_type', 'mixto')
            type_counts[mtype] = type_counts.get(mtype, 0) + 1

        if type_counts:
```

```
            return max(type_counts.items(), key=lambda x: x[1])[0]
        return 'mixto'

    @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_r_hat")
    def _calculate_r_hat(self, chains: list[list[dict[str, Any]]]) -> float:
        """Calcula Gelman-Rubin R-hat para diagnÃ³stico de convergencia"""
        if len(chains) < 2:
            return 1.0

        # Extraer coherence de cada cadena
        chain_means = []
        chain_vars = []

        for chain in chains:
            coherences = [s.get('coherence', 0.5) for s in chain]
            if len(coherences) > 0:
                chain_means.append(np.mean(coherences))
                chain_vars.append(np.var(coherences, ddof=1))

        if len(chain_means) < 2:
            return 1.0

        # Between-chain variance (B)
        n = len(chains[0])  # samples per chain
        B = np.var(chain_means, ddof=1) * n

        # Within-chain variance (W)
        W = np.mean(chain_vars)

        # R-hat estimator
        if W > 0:
            var_plus = ((n - 1) / n) * W + (1 / n) * B
            r_hat = np.sqrt(var_plus / W)
        else:
            r_hat = 1.0 # Refactored

        return float(r_hat)

    @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess")
    def _calculate_ess(self, samples: list[dict[str, Any]]) -> float:
        """Calcula Effective Sample Size (simplificado)"""
        n = len(samples)

        # Estimar autocorrelaciÃ³n
        coherences = np.array([s.get('coherence', 0.5) for s in samples])

        if len(coherences) < 2:
            return n

        # Lag-1 autocorrelation
        mean_coh = np.mean(coherences)
        var_coh = np.var(coherences)

        if var_coh > 0:
            lag1_autocorr = np.mean(
                (coherences[:-1] - mean_coh) * (coherences[1:] - mean_coh)
```

```
            ) / var_coh
        else:
            lag1_autocorr = 0.0 # Refactored

        # ESS approximation
        ess = n / (1 + 2 * max(0, lag1_autocorr))
        return float(ess)

    def posterior_predictive_check(
        self,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, Any]:
        """
        PROMPT II-2: Posterior Predictive Checks + Ablation

        Genera datos simulados desde posterior y compara con observados.
        Realiza ablation de pasos de secuencia.

        Args:
            posterior_samples: Samples del posterior MCMC
            observed_data: Datos observados reales

        Returns:
            Dict con ppd_p_value, distance_metric, ablation_curve, criteria_met
        """
        self.logger.info("Running posterior predictive checks...")

        # 1. Generar datos predictivos desde posterior
        n_ppd_samples = min(100, len(posterior_samples))
        ppd_samples = []

        for _i in range(n_ppd_samples):
            sample_idx = np.random.randint(0, len(posterior_samples))
            posterior_sample = posterior_samples[sample_idx]

            # Simular coherence desde distribuciÃ³n posterior
            simulated_coherence = posterior_sample.get('coherence', 0.5) + np.random.normal(0, 0.05)
            simulated_coherence = np.clip(simulated_coherence, 0.0, 1.0)
            ppd_samples.append(simulated_coherence)

        ppd_samples = np.array(ppd_samples)

        # 2. Comparar con observado usando KS test
        observed_coherence = observed_data.get('coherence', 0.5)

        # KS test: comparar distribuciÃ³n PPD con punto observado
        from scipy.stats import kstest
        ks_stat, ppd_p_value = kstest(ppd_samples, lambda x: 0 if x < observed_coherence else 1)
        ppd_p_value = float(ppd_p_value)

        # 3. Ablation de secuencia
        ablation_curve = self._ablation_analysis(posterior_samples, observed_data)

        # 4. Verificar criterios
        ppd_ok = 0.1 <= ppd_p_value <= 0.9
```

```
        ablation_ok = all(delta >= -0.05 for delta in ablation_curve.values())  # Tolerancia -5%

        criteria_met = {
            ’ppd_p_value_ok’: ppd_ok,
            ’ablation_ok’: ablation_ok
        }

        # RecomendaciÃ³n
        if ppd_ok and ablation_ok:
            recommendation = ’accept’
        else:
            recommendation = ’rebaja_posterior’
            self.logger.warning(f"PPC failed: ppd_p={ppd_p_value:.3f}, ablation_ok={ablation_ok}")

        return {
            ’ppd_p_value’: ppd_p_value,
            ’ppd_samples_mean’: float(np.mean(ppd_samples)),
            ’ppd_samples_std’: float(np.std(ppd_samples)),
            ’distance_metric’: ’KS’,
            ’ks_statistic’: float(ks_stat),
            ’ablation_curve’: ablation_curve,
            ’criteria_met’: criteria_met,
            ’recommendation’: recommendation
        }

    def _ablation_analysis(
        self,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, float]:
        """Mide caÃda en coherence al quitar pasos de secuencia"""
        baseline_coherence = np.mean([s.get(’coherence’, 0.5) for s in posterior_samples])

        # Simular ablaciÃ³n de pasos clave
        # En prÃ¡ctica real, esto requerirÃa re-ejecutar modelo sin ciertos steps
        ablation_deltas = {
            ’remove_step_diagnostic’: baseline_coherence - (baseline_coherence * 0.95),  # -5%
            ’remove_step_planning’: baseline_coherence - (baseline_coherence * 0.85),    # -15%
            ’remove_step_execution’: baseline_coherence - (baseline_coherence * 0.90),   # -10%
            ’remove_step_monitoring’: baseline_coherence - (baseline_coherence * 0.97)   # -3%
        }

        return ablation_deltas

    def verify_conditional_independence(
        self,
        dag: nx.DiGraph,
        independence_tests: list[tuple[str, str, list[str]]] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT II-3: Independencias y parsimonia

        Verifica d-separaciones implicadas por el DAG.
        Calcula Î\224WAIC entre modelo jerÃ¡rquico vs. nulo.

        Args:
```

```
            dag: NetworkX DiGraph del modelo causal
            independence_tests: Lista de tuplas (X, Y, Z) para test X â\212¥ Y | Z

        Returns:
            Dict con independence_tests, delta_waic, model_preference, criteria_met
        """
        self.logger.info("Verifying conditional independencies...")

        # 1. Tests de independencia (d-separaciÃ³n)
        test_results = []

        if independence_tests is None:
            # Generar tests automÃ¡ticamente si no se proveen
            independence_tests = self._generate_independence_tests(dag)

        for x, y, z_set in independence_tests:
            try:
                # Verificar d-separaciÃ³n en DAG
                is_independent = nx.d_separated(dag, {x}, {y}, set(z_set))
                test_results.append({
                    'test': f"{x} â\212¥ {y} | {{{', '.join(z_set)}}}",
                    'x': x,
                    'y': y,
                    'z': z_set,
                    'passed': is_independent
                })
            except Exception as e:
                self.logger.warning(f"Independence test failed: {x} â\212¥ {y} | {z_set} - {e}")
                test_results.append({
                    'test': f"{x} â\212¥ {y} | {{{', '.join(z_set)}}}",
                    'x': x,
                    'y': y,
                    'z': z_set,
                    'passed': False,
                    'error': str(e)
                })

        tests_passed = sum(1 for t in test_results if t['passed'])

        # 2. Calcular Î\224WAIC (simplificado)
        # En prÃ¡ctica real: usar librerÃa como arviz para WAIC calculation
        delta_waic = self._calculate_waic_difference(dag)

        # 3. Verificar criterios
        independence_ok = tests_passed >= 2
        waic_ok = delta_waic <= -2.0

        # 4. Preferencia de modelo
        if independence_ok and waic_ok:
            model_preference = 'hierarchical'
        elif not waic_ok:
            model_preference = 'inconclusive'
        else:
            model_preference = 'null'

        criteria_met = {
```

```
            'independence_ok': independence_ok,
            'waic_ok': waic_ok
        }

        return {
            'independence_tests': test_results,
            'tests_passed': tests_passed,
            'tests_total': len(test_results),
            'delta_waic': float(delta_waic),
            'model_preference': model_preference,
            'criteria_met': criteria_met
        }

    def _generate_independence_tests(
        self,
        dag: nx.DiGraph,
        n_tests: int = 3
    ) -> list[tuple[str, str, list[str]]]:
        """Genera tests de independencia automÃ¡ticamente desde DAG"""
        tests = []
        nodes = list(dag.nodes())

        if len(nodes) < 3:
            return tests

        # Generar tests de forma heurÃstica
        for _ in range(min(n_tests, len(nodes) - 2)):
            # Seleccionar nodos aleatorios
            x, y = np.random.choice(nodes, size=2, replace=False)

            # Z: padres comunes o mediadores
            z_candidates = set(dag.predecessors(x)) | set(dag.predecessors(y))
            z_set = list(z_candidates)[:2]  # MÃ¡ximo 2 nodos en conditioning set

            if x != y:
                tests.append((x, y, z_set))

        return tests

    @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_waic_difference")
    def _calculate_waic_difference(self, dag: nx.DiGraph) -> float:
        """
        Calcula Î\224WAIC = WAIC_hierarchical - WAIC_null (simplificado)

        En producciÃ³n: usar arviz.waic() con trace real de PyMC/Stan
        """
        # HeurÃstica: modelos jerÃ¡rquicos con mÃ¡s estructura (edges) son preferidos
        n_edges = dag.number_of_edges()
        dag.number_of_nodes()

        # PenalizaciÃ³n por complejidad
        complexity_penalty = n_edges * 0.5

        # WAIC aproximado
        waic_hierarchical = -50.0 - n_edges * 2  # Mejor fit con mÃ¡s estructura
        waic_null = -45.0  # Modelo nulo sin estructura
```

```

        delta_waic = waic_hierarchical - waic_null + complexity_penalty

        return delta_waic

# ============================================================================
# AGUJA III: AUDITOR CONTRAFACTUAL BAYESIANO
# ============================================================================

class BayesianCounterfactualAuditor:
    """
    AGUJA III - Auditor Contrafactual con SCM y do-calculus

    PROMPT III-1: ConstrucciÃ³n de SCM y queries gemelas
    Construye SCM={DAG, f_i} y responde omission_impact, sufficiency_test, necessity_test.

    PROMPT III-2: Riesgo sistÃ©mico y priorizaciÃ³n
    Agrega riesgos, propaga incertidumbre, calcula priority.

    PROMPT III-3: RefutaciÃ³n, negativos y cordura do(.)
    Ejecuta controles negativos, pruebas placebo, sanity checks.

    QUALITY CRITERIA:
    - Consistencia de signos factual/contrafactual
    - effect_stability: Î\224effect â\211¤ 0.15) al variar priors Â±10%
    - negative_controls: mediana |efecto| â\211¤ 0.05)
    - sanity_violations: 0
    """

    def __init__(self) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.scm: dict[str, Any] | None = None

    def construct_scm(
        self,
        dag: nx.DiGraph,
        structural_equations: dict[str, callable] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-1: ConstrucciÃ³n de SCM

        Construye SCM = {DAG, f_i} desde grafo y ecuaciones estructurales.

        Args:
            dag: NetworkX DiGraph (debe ser acÃ-clico)
            structural_equations: Dict {node: function} para f_i

        Returns:
            SCM con DAG validado y funciones estructurales

        Raises:
            ValueError: Si DAG no es acÃ-clico
        """
        self.logger.info(f"Constructing SCM with {dag.number_of_nodes()} nodes, {dag.number_of_edges()} edges")

        # 1. Validar que DAG es acÃ-clico
```

```
        if not nx.is_directed_acyclic_graph(dag):
            raise ValueError("DAG must be acyclic for SCM construction. Use cycle detection first.")

        # 2. Crear ecuaciones por defecto si no se proveen
        if structural_equations is None:
            structural_equations = self._create_default_equations(dag)
            self.logger.info(f"Created {len(structural_equations)} default structural equations")

        # 3. Construir SCM
        scm = {
            'dag': dag,
            'equations': structural_equations,
            'nodes': list(dag.nodes()),
            'edges': list(dag.edges()),
            'topological_order': list(nx.topological_sort(dag))
        }

        self.scm = scm
        self.logger.info("â\234\223 SCM constructed successfully")
        return scm

    @calibrated_method("farfan_core.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations")
    def _create_default_equations(self, dag: nx.DiGraph) -> dict[str, callable]:
        """Crea ecuaciones estructurales lineales por defecto"""
        equations = {}

        for node in dag.nodes():
            parents = list(dag.predecessors(node))

            if not parents:
                # Nodo raÃz: variable exÃ³gena U
                def root_eq(noise=0.0, node_name=node):
                    return 0.5 + noise  # Prior neutral + ruido
                equations[node] = root_eq
            else:
                # Nodo con padres: funciÃ³n lineal
                def child_eq(parent_values, noise=0.0, node_name=node, n_parents=len(parents)):
                    if isinstance(parent_values, dict):
                        return sum(parent_values.values()) / max(n_parents, 1) + noise
                    return 0.5 + noise
                equations[node] = child_eq

        return equations

    def counterfactual_query(
        self,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-1: Queries gemelas (omission, sufficiency, necessity)

        EvalÃºa:
        - Factual: P(Y | evidence)
        - Counterfactual: P(Y | do(X=x), evidence)
```

```
        - Causal effect, sufficiency, necessity

        Args:
            intervention: {nodo: valor} para do(.) operation
            target: Nodo objetivo Y
            evidence: Evidencia observada (opcional)

        Returns:
            Dict con p_factual, p_counterfactual, causal_effect, is_sufficient, is_necessary
        """
        if self.scm is None:
            raise ValueError("SCM must be constructed first. Call construct_scm().")

        evidence = evidence or {}

        self.logger.debug(f"Counterfactual query: intervention={intervention}, target={target}")

        # 1. Factual: P(Y | evidence)
        p_factual = self._evaluate_factual(target, evidence)

        # 2. Counterfactual: P(Y | do(X=x), evidence)
        p_counterfactual = self._evaluate_counterfactual(target, intervention, evidence)

        # 3. Causal effect
        causal_effect = p_counterfactual - p_factual

        # 4. Sufficiency test: Â¿do(X=1) â\206\222 Y=1?
        intervention_node = list(intervention.keys())[0] if intervention else None
        if intervention_node:
            p_y_given_do_x1 = self._evaluate_counterfactual(target, {intervention_node: 1.0}, {})
            is_sufficient = p_y_given_do_x1 > 0.7
        else:
            is_sufficient = False

        # 5. Necessity test: Â¿do(X=0) â\206\222 Y=0?
        if intervention_node:
            p_y_given_do_x0 = self._evaluate_counterfactual(target, {intervention_node: 0.0}, {})
            is_necessary = p_y_given_do_x0 < 0.3
        else:
            is_necessary = False

        # 6. Consistencia de signos
        signs_consistent = (
            (causal_effect >= 0 and p_counterfactual >= p_factual) or
            (causal_effect < 0 and p_counterfactual < p_factual)
        )

        # 7. Effect stability
        stability = self._test_effect_stability(intervention, target, evidence)

        return {
            'p_factual': float(np.clip(p_factual, 0.0, 1.0)),
            'p_counterfactual': float(np.clip(p_counterfactual, 0.0, 1.0)),
            'causal_effect': float(causal_effect),
            'is_sufficient': is_sufficient,
            'is_necessary': is_necessary,
```

```
            'signs_consistent': signs_consistent,
            'effect_stability': float(stability),
            'effect_stable': stability <= 0.15
        }

    def _evaluate_factual(
        self,
        target: str,
        evidence: dict[str, float]
    ) -> float:
        """EvalÃºa P(target | evidence) propagando hacia adelante en DAG"""
        if target in evidence:
            return evidence[target]

        dag = self.scm['dag']
        equations = self.scm['equations']
        topological_order = self.scm['topological_order']

        # Evaluar nodos en orden topolÃ³gico
        computed_values = evidence.copy()

        for node in topological_order:
            if node in computed_values:
                continue

            parents = list(dag.predecessors(node))

            if not parents:
                # Nodo raÃz
                computed_values[node] = equations[node](noise=0.0)
            else:
                # Evaluar padres primero
                parent_values = {}
                for parent in parents:
                    if parent not in computed_values:
                        computed_values[parent] = self._evaluate_factual(parent, evidence)
                    parent_values[parent] = computed_values[parent]

                # Aplicar ecuaciÃ³n estructural
                try:
                    computed_values[node] = equations[node](parent_values, noise=0.0)
                except:
                    # Fallback
                    computed_values[node] = sum(parent_values.values()) / max(len(parent_values), 1)

        return float(np.clip(computed_values.get(target, 0.5), 0.0, 1.0))

    def _evaluate_counterfactual(
        self,
        target: str,
        intervention: dict[str, float],
        evidence: dict[str, float]
    ) -> float:
        """EvalÃºa P(target | do(intervention), evidence) con DAG mutilado"""
        # Crear DAG mutilado: quitar aristas hacia nodos intervenidos
        dag_mutilated = self.scm['dag'].copy()
```

```
        for node in intervention:
            in_edges = list(dag_mutilated.in_edges(node))
            dag_mutilated.remove_edges_from(in_edges)

        # Guardar SCM original
        original_scm = self.scm.copy()

        # Crear SCM mutilado temporalmente
        self.scm = {
            'dag': dag_mutilated,
            'equations': self.scm['equations'],
            'nodes': self.scm['nodes'],
            'edges': list(dag_mutilated.edges()),
            'topological_order': list(nx.topological_sort(dag_mutilated))
        }

        # Combinar evidence con intervention (intervention tiene prioridad)
        combined_evidence = {**evidence, **intervention}

        # Evaluar en SCM mutilado
        result = self._evaluate_factual(target, combined_evidence)

        # Restaurar SCM original
        self.scm = original_scm

        return result

    def _test_effect_stability(
        self,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None,
        n_perturbations: int = 5
    ) -> float:
        """Testa estabilidad al variar priors/ecuaciones Â±10%"""
        evidence = evidence or {}

        # Efecto baseline
        baseline_result = self.counterfactual_query(intervention, target, evidence)
        baseline_effect = baseline_result['causal_effect']

        # Perturbar y medir variaciÃ³n
        perturbed_effects = []

        for _ in range(n_perturbations):
            perturbation_factor = np.random.uniform(0.9, 1.1)  # Â±10%

            # Perturbar valores de evidencia
            perturbed_evidence = {
                k: v * perturbation_factor for k, v in evidence.items()
            }

            # Re-evaluar
            try:
                result = self.counterfactual_query(intervention, target, perturbed_evidence)
```

```
                perturbed_effects.append(result['causal_effect'])
            except:
                perturbed_effects.append(baseline_effect)

        # MÃ¡xima variaciÃ³n
        max_variation = max(abs(e - baseline_effect) for e in perturbed_effects) if perturbed_effects else 0.0

        return max_variation

    def aggregate_risk_and_prioritize(
        self,
        omission_score: float,
        insufficiency_score: float,
        unnecessity_score: float,
        causal_effect: float,
        feasibility: float = 0.8,
        cost: float = 1.0
    ) -> dict[str, Any]:
        """
        PROMPT III-2: Riesgo sistÃ©mico y priorizaciÃ³n con incertidumbre

        FÃ³rmulas:
        - risk = 0.50Â•omission + 0.35Â•insufficiency + 0.15Â•unnecessity
        - priority = |effect|Â•feasibility/(cost+Îµ)Â•(1â\210\222uncertainty)

        Args:
            omission_score: Riesgo de omisiÃ³n de mecanismo [0,1]
            insufficiency_score: Insuficiencia del mecanismo [0,1]
            unnecessity_score: Mecanismo innecesario [0,1]
            causal_effect: Efecto causal estimado
            feasibility: Factibilidad de intervenciÃ³n [0,1]
            cost: Costo relativo (>0)

        Returns:
            Dict con risk_score, success_probability, priority, recommendations
        """
        # 1. Componentes de riesgo
        risk_components = {
            'omission': float(np.clip(omission_score, 0.0, 1.0)),
            'insufficiency': float(np.clip(insufficiency_score, 0.0, 1.0)),
            'unnecessity': float(np.clip(unnecessity_score, 0.0, 1.0))
        }

        # 2. Riesgo agregado
        risk_score = (
            0.50 * risk_components['omission'] +
            0.35 * risk_components['insufficiency'] +
            0.15 * risk_components['unnecessity']
        )
        risk_score = float(np.clip(risk_score, 0.0, 1.0))

        # 3. Success probability con incertidumbre
        success_mean = 1.0 - risk_score

        # Incertidumbre: mayor riesgo â\206\222 mayor uncertainty
        success_std = 0.05 + 0.10 * risk_score  # Entre 5% y 15%
```

```
        # CI95 para success
        ci95_low = max(0.0, success_mean - 1.96 * success_std)
        ci95_high = min(1.0, success_mean + 1.96 * success_std)

        success_probability = {
            'mean': float(success_mean),
            'std': float(success_std),
            'CI95': (float(ci95_low), float(ci95_high))
        }

        # 4. Prioridad
        uncertainty = success_std
        epsilon = 1e-6

        priority = (
            abs(causal_effect) *
            feasibility /
            (cost + epsilon) *
            (1.0 - uncertainty)
        )
        priority = float(priority)

        # 5. Recomendaciones ordenadas
        recommendations = []

        if risk_score > 0.7:
            recommendations.append("CRITICAL_RISK: Immediate intervention required")
        elif risk_score > 0.4:
            recommendations.append("MEDIUM_RISK: Close monitoring required")
        else:
            recommendations.append("LOW_RISK: Routine surveillance")

        if risk_components['omission'] > 0.6:
            recommendations.append("HIGH_OMISSION_RISK: Key mechanism may be missing")

        if risk_components['insufficiency'] > 0.5:
            recommendations.append("INSUFFICIENCY_DETECTED: Mechanism alone insufficient")

        if priority > 0.5:
            recommendations.append("HIGH_PRIORITY: Optimal intervention candidate")
        elif priority < 0.2:
            recommendations.append("LOW_PRIORITY: Consider alternative interventions")

        # 6. Verificar criterios de calidad
        ci95_valid = 0.0 <= ci95_low <= ci95_high <= 1.0
        priority_monotonic = priority >= 0
        risk_in_range = 0.0 <= risk_score <= 1.0

        criteria_met = {
            'ci95_valid': ci95_valid,
            'priority_monotonic': priority_monotonic,
            'risk_in_range': risk_in_range
        }

        return {
```

```
            'risk_components': risk_components,
            'risk_score': risk_score,
            'success_probability': success_probability,
            'priority': priority,
            'recommendations': sorted(recommendations, reverse=True),
            'criteria_met': criteria_met
        }

    def refutation_and_sanity_checks(
        self,
        dag: nx.DiGraph,
        target: str,
        treatment: str,
        confounders: list[str] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-3: RefutaciÃ³n, negativos y cordura do(.)

        Ejecuta:
        1. Controles negativos: nodos irrelevantes â\206\222 |efecto| â\211¤ 0.05)
        2. Pruebas placebo: permuta edges no causales
        3. Sanity checks: aÃ±adir cofactores no reduce P(Y|do(X=1))

        Args:
            dag: Grafo causal
            target: Nodo objetivo Y
            treatment: Nodo de tratamiento X
            confounders: Lista de cofactores

        Returns:
            Dict con negative_controls, placebo_effect, sanity_violations, recommendation
        """
        confounders = confounders or []

        self.logger.info("Running refutation and sanity checks...")

        # 1. CONTROLES NEGATIVOS: nodos irrelevantes
        irrelevant_nodes = [
            n for n in dag.nodes()
            if n not in (target, treatment) and not nx.has_path(dag, n, target)
        ]

        negative_effects = []
        for node in irrelevant_nodes[:5]:  # MÃ¡ximo 5 controles
            try:
                intervention = {node: 1.0}
                result = self.counterfactual_query(intervention, target, {})
                effect = abs(result['causal_effect'])
                negative_effects.append(effect)
            except Exception as e:
                self.logger.warning(f"Negative control failed for {node}: {e}")

        median_negative_effect = float(np.median(negative_effects)) if negative_effects else 0.0
        negative_controls_ok = median_negative_effect <= 0.05

        # 2. PRUEBA PLACEBO: permuta edges no causales
```

```
        placebo_dag = dag.copy()
        non_causal_edges = [
            (u, v) for u, v in dag.edges()
            if u != treatment and v != target
        ]

        placebo_effect = 0.0 # Refactored
        if non_causal_edges:
            # Permutar una arista
            edge_to_remove = non_causal_edges[0]
            placebo_dag.remove_edge(*edge_to_remove)

            # Medir efecto en DAG permutado
            scm_backup = self.scm
            try:
                self.construct_scm(placebo_dag)
                result = self.counterfactual_query({treatment: 1.0}, target, {})
                placebo_effect = abs(result['causal_effect'])
            except Exception as e:
                self.logger.warning(f"Placebo test failed: {e}")
            finally:
                self.scm = scm_backup

        placebo_ok = placebo_effect <= 0.05

        # 3. SANITY CHECKS: aÃ±adir cofactores activos no debe reducir P(Y|do(X=1))
        sanity_violations = []

        # Baseline: do(X=1)
        try:
            baseline_result = self.counterfactual_query({treatment: 1.0}, target, {})
            baseline_p = baseline_result['p_counterfactual']

            # Con cofactores
            for confounder in confounders[:2]:  # MÃ¡ximo 2
                if confounder in dag.nodes():
                    result_with_conf = self.counterfactual_query(
                        {treatment: 1.0},
                        target,
                        {confounder: 1.0}
                    )
                    p_with_conf = result_with_conf['p_counterfactual']

                    # Verificar que no reduce significativamente
                    if p_with_conf < baseline_p - 0.10:
                        sanity_violations.append({
                            'confounder': confounder,
                            'baseline_p': float(baseline_p),
                            'p_with_confounder': float(p_with_conf),
                            'violation': f"Adding {confounder} reduced P(Y|do(X)) by {baseline_p - p_with_conf:.3f}"
                        })
        except Exception as e:
            self.logger.error(f"Sanity checks failed: {e}")

        sanity_ok = len(sanity_violations) == 0
```

```
        # 4. DECISIÃ\223N FINAL
        all_checks_passed = negative_controls_ok and placebo_ok and sanity_ok

        if not all_checks_passed:
            recommendation = "DEGRADE_ALL: Require DAG revision - observaciÃ³n prioritaria"
            self.logger.error(recommendation)
        else:
            recommendation = "ACCEPT: All refutation tests passed"
            self.logger.info(recommendation)

        return {
            'negative_controls': {
                'effects': [float(e) for e in negative_effects],
                'median': median_negative_effect,
                'passed': negative_controls_ok,
                'criterion': 'â\211¤ 0.05)'
            },
            'placebo_effect': {
                'effect': float(placebo_effect),
                'passed': placebo_ok,
                'criterion': 'â\211\210 0'
            },
            'sanity_violations': sanity_violations,
            'sanity_passed': sanity_ok,
            'all_checks_passed': all_checks_passed,
            'recommendation': recommendation
        }

def main() -> int:
    """CLI entry point"""
    parser = argparse.ArgumentParser(
        description="CDAF v2.0 - Framework de DeconstrucciÃ³n y AuditorÃa Causal",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Ejemplo de uso:
  python cdaf_framework.py documento.pdf --output-dir resultados/ --policy-code P1

ConfiguraciÃ³n:
  El framework busca config.yaml en el directorio actual.
  Use --config-file para especificar una ruta alternativa.
        """
    )

    parser.add_argument(
        "pdf_path",
        type=Path,
        help="Ruta al archivo PDF del Plan de Desarrollo Territorial"
    )

    parser.add_argument(
        "--output-dir",
        type=Path,
        default=Path("resultados_analisis"),
        help="Directorio de salida para los artefactos (default: resultados_analisis/)"
    )

```

```
    parser.add_argument(
        "--policy-code",
        type=str,
        required=True,
        help="CÃ³digo de polÃ-tica para nombrar los artefactos (ej: P1, PDT_2024)"
    )

    parser.add_argument(
        "--config-file",
        type=Path,
        default=Path(DEFAULT_CONFIG_FILE),
        help=f"Ruta al archivo de configuraciÃ³n YAML (default: {DEFAULT_CONFIG_FILE})"
    )

    parser.add_argument(
        "--log-level",
        choices=["DEBUG", "INFO", "WARNING", "ERROR"],
        default="INFO",
        help="Nivel de logging (default: INFO)"
    )

    parser.add_argument(
        "--pdet",
        action="store_true",
        help="Indica si el municipio es PDET (activa validaciÃ³n especial)"
    )

    args = parser.parse_args()

    # Validate inputs
    if not args.pdf_path.exists():
        print(f"ERROR: Archivo PDF no encontrado: {args.pdf_path}")
        return 1

    # Initialize framework
    try:
        framework = CDAFFramework(args.config_file, args.output_dir, args.log_level)

        # Configure PDET if specified
        if args.pdet and framework.dnp_validator:
            framework.dnp_validator.es_municipio_pdet = True
            framework.logger.info("Modo PDET activado - ValidaciÃ³n especial habilitada")
    except Exception as e:
        print(f"ERROR: No se pudo inicializar el framework: {e}")
        return 1

    # Process document
    success = framework.process_document(args.pdf_path, args.policy_code)

    return 0 if success else 1

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

class DerekBeachProducer:
```

```
    """
    Producer wrapper for Derek Beach causal analysis with registry exposure

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

    Version: 1.0).0
    Producer Type: Causal Mechanism Analysis
    """

    def __init__(self) -> None:
        """Initialize producer"""
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.info("DerekBeachProducer initialized")

    # ========================================================================
    # EVIDENTIAL TESTS API
    # ========================================================================

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.classify_test_type")
    def classify_test_type(self, necessity: float, sufficiency: float) -> TestType:
        """Classify evidential test type based on necessity and sufficiency"""
        return BeachEvidentialTest.classify_test(necessity, sufficiency)

    def apply_test_logic(
        self,
        test_type: TestType,
        evidence_found: bool,
        prior: float,
        bayes_factor: float
    ) -> tuple[float, str]:
        """Apply Beach test-specific logic to Bayesian updating"""
        return BeachEvidentialTest.apply_test_logic(
            test_type, evidence_found, prior, bayes_factor
        )

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_hoop_test")
    def is_hoop_test(self, test_type: TestType) -> bool:
        """Check if test is hoop test"""
        return test_type == "hoop_test"

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_smoking_gun")
    def is_smoking_gun(self, test_type: TestType) -> bool:
        """Check if test is smoking gun"""
        return test_type == "smoking_gun"

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_doubly_decisive")
    def is_doubly_decisive(self, test_type: TestType) -> bool:
        """Check if test is doubly decisive"""
        return test_type == "doubly_decisive"

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_straw_in_wind")
    def is_straw_in_wind(self, test_type: TestType) -> bool:
        """Check if test is straw in wind"""
        return test_type == "straw_in_wind"
```

```
    # ========================================================================
    # HIERARCHICAL GENERATIVE MODEL API
    # ========================================================================

    def create_hierarchical_model(
        self,
        mechanism_priors: dict[str, float] | None = None
    ) -> HierarchicalGenerativeModel:
        """Create hierarchical generative model"""
        return HierarchicalGenerativeModel(mechanism_priors)

    def infer_mechanism_posterior(
        self,
        model: HierarchicalGenerativeModel,
        observations: dict[str, Any],
        n_iter: int = 500,
        burn_in: int = 100,
        n_chains: int = 2
    ) -> dict[str, Any]:
        """Infer mechanism posterior using MCMC"""
        return model.infer_mechanism_posterior(
            observations, n_iter, burn_in, n_chains
        )

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_type_posterior")
    def get_type_posterior(self, inference: dict[str, Any]) -> dict[str, float]:
        """Extract type posterior from inference"""
        return inference.get("type_posterior", {})

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_sequence_mode")
    def get_sequence_mode(self, inference: dict[str, Any]) -> str:
        """Extract sequence mode from inference"""
        return inference.get("sequence_mode", "")

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_coherence_score")
    def get_coherence_score(self, inference: dict[str, Any]) -> float:
        """Extract coherence score from inference"""
        return inference.get("coherence_score", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_r_hat")
    def get_r_hat(self, inference: dict[str, Any]) -> float:
        """Extract R-hat convergence diagnostic"""
        return inference.get("R_hat", 1.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ess")
    def get_ess(self, inference: dict[str, Any]) -> float:
        """Extract effective sample size"""
        return inference.get("ESS", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_inference_uncertain")
    def is_inference_uncertain(self, inference: dict[str, Any]) -> bool:
        """Check if inference has high uncertainty"""
        return inference.get("is_uncertain", False)

    # ========================================================================
    # POSTERIOR PREDICTIVE CHECKS API
```

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

    def posterior_predictive_check(
        self,
        model: HierarchicalGenerativeModel,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, Any]:
        """Run posterior predictive checks"""
        return model.posterior_predictive_check(posterior_samples, observed_data)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ppd_p_value")
    def get_ppd_p_value(self, ppc: dict[str, Any]) -> float:
        """Extract posterior predictive p-value"""
        return ppc.get("ppd_p_value", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ablation_curve")
    def get_ablation_curve(self, ppc: dict[str, Any]) -> dict[str, float]:
        """Extract ablation curve from PPC"""
        return ppc.get("ablation_curve", {})

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ppc_recommendation")
    def get_ppc_recommendation(self, ppc: dict[str, Any]) -> str:
        """Extract recommendation from PPC"""
        return ppc.get("recommendation", "")

    # ========================================================================
    # CONDITIONAL INDEPENDENCE API
    # ========================================================================

    def verify_conditional_independence(
        self,
        model: HierarchicalGenerativeModel,
        dag: nx.DiGraph,
        independence_tests: list[tuple[str, str, list[str]]] | None = None
    ) -> dict[str, Any]:
        """Verify conditional independencies in DAG"""
        return model.verify_conditional_independence(dag, independence_tests)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_independence_tests")
    def get_independence_tests(self, verification: dict[str, Any]) -> list[dict[str, Any]]:
        """Extract independence tests from verification"""
        return verification.get("independence_tests", [])

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_delta_waic")
    def get_delta_waic(self, verification: dict[str, Any]) -> float:
        """Extract delta WAIC from verification"""
        return verification.get("delta_waic", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_model_preference")
    def get_model_preference(self, verification: dict[str, Any]) -> str:
        """Extract model preference from verification"""
        return verification.get("model_preference", "inconclusive")

    # ========================================================================
    # COUNTERFACTUAL AUDITOR API
```

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

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.create_auditor")
    def create_auditor(self) -> BayesianCounterfactualAuditor:
        """Create Bayesian counterfactual auditor"""
        return BayesianCounterfactualAuditor()

    def construct_scm(
        self,
        auditor: BayesianCounterfactualAuditor,
        dag: nx.DiGraph,
        structural_equations: dict[str, callable] | None = None
    ) -> dict[str, Any]:
        """Construct structural causal model"""
        return auditor.construct_scm(dag, structural_equations)

    def counterfactual_query(
        self,
        auditor: BayesianCounterfactualAuditor,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None = None
    ) -> dict[str, Any]:
        """Execute counterfactual query"""
        return auditor.counterfactual_query(intervention, target, evidence)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_causal_effect")
    def get_causal_effect(self, query: dict[str, Any]) -> float:
        """Extract causal effect from query"""
        return query.get("causal_effect", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_sufficient")
    def is_sufficient(self, query: dict[str, Any]) -> bool:
        """Check if mechanism is sufficient"""
        return query.get("is_sufficient", False)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_necessary")
    def is_necessary(self, query: dict[str, Any]) -> bool:
        """Check if mechanism is necessary"""
        return query.get("is_necessary", False)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_effect_stable")
    def is_effect_stable(self, query: dict[str, Any]) -> bool:
        """Check if effect is stable"""
        return query.get("effect_stable", False)

    # ========================================================================
    # RISK AGGREGATION API
    # ========================================================================

    def aggregate_risk(
        self,
        auditor: BayesianCounterfactualAuditor,
        omission_score: float,
        insufficiency_score: float,
        unnecessity_score: float,
```

```
        causal_effect: float,
        feasibility: float = 0.8,
        cost: float = 1.0
    ) -> dict[str, Any]:
        """Aggregate risk and calculate priority"""
        return auditor.aggregate_risk_and_prioritize(
            omission_score,
            insufficiency_score,
            unnecessity_score,
            causal_effect,
            feasibility,
            cost
        )

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_risk_score")
    def get_risk_score(self, aggregation: dict[str, Any]) -> float:
        """Extract risk score from aggregation"""
        return aggregation.get("risk_score", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_success_probability")
    def get_success_probability(self, aggregation: dict[str, Any]) -> dict[str, float]:
        """Extract success probability from aggregation"""
        return aggregation.get("success_probability", {})

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_priority")
    def get_priority(self, aggregation: dict[str, Any]) -> float:
        """Extract priority from aggregation"""
        return aggregation.get("priority", 0.0)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_recommendations")
    def get_recommendations(self, aggregation: dict[str, Any]) -> list[str]:
        """Extract recommendations from aggregation"""
        return aggregation.get("recommendations", [])

    # ========================================================================
    # REFUTATION API
    # ========================================================================

    def refutation_checks(
        self,
        auditor: BayesianCounterfactualAuditor,
        dag: nx.DiGraph,
        target: str,
        treatment: str,
        confounders: list[str] | None = None
    ) -> dict[str, Any]:
        """Execute refutation and sanity checks"""
        return auditor.refutation_and_sanity_checks(
            dag, target, treatment, confounders
        )

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_negative_controls")
    def get_negative_controls(self, refutation: dict[str, Any]) -> dict[str, Any]:
        """Extract negative controls from refutation"""
        return refutation.get("negative_controls", {})

```

```
    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_placebo_effect")
    def get_placebo_effect(self, refutation: dict[str, Any]) -> dict[str, Any]:
        """Extract placebo effect from refutation"""
        return refutation.get("placebo_effect", {})

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_sanity_violations")
    def get_sanity_violations(self, refutation: dict[str, Any]) -> list[dict[str, Any]]:
        """Extract sanity violations from refutation"""
        return refutation.get("sanity_violations", [])

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.all_checks_passed")
    def all_checks_passed(self, refutation: dict[str, Any]) -> bool:
        """Check if all refutation checks passed"""
        return refutation.get("all_checks_passed", False)

    @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_refutation_recommendation")
    def get_refutation_recommendation(self, refutation: dict[str, Any]) -> str:
        """Extract recommendation from refutation"""
        return refutation.get("recommendation", "")



================================================================================
FILE: src/farfan_pipeline/analysis/enhance_recommendation_rules.py
================================================================================

#!/usr/bin/env python3
"""
Script to enhance recommendation rules with 7 advanced features:
1. Template parameterization
2. Rule execution logic
3. Measurable indicators
4. Unambiguous time horizons
5. Testable verification
6. Cost tracking
7. Authority mapping
"""

import copy
import json
from pathlib import Path
from typing import Any
from farfan_pipeline.core.calibration.decorators import calibrated_method


def enhance_template(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 1: Eliminate Hardcoded Template Strings
    Replace with template_id and template_params
    """
    template = rule.get('template', {})
    rule_id = rule.get('rule_id', '')
    level = rule.get('level', '')

    # Extract parameters from template strings
    template_params = {}
```

```

    if level == 'MICRO':
        when = rule.get('when', {})
        template_params = {
            'pa_id': when.get('pa_id', 'PA01'),
            'dim_id': when.get('dim_id', 'DIM01'),
            'question_id': 'Q001'  # Would be derived from context
        }
    elif level == 'MESO':
        when = rule.get('when', {})
        template_params = {
            'cluster_id': when.get('cluster_id', 'CL01')
        }

    # Create enhanced template with ID
    enhanced_template = copy.deepcopy(template)
    enhanced_template['template_id'] = f"TPL-{rule_id}"
    enhanced_template['template_params'] = template_params

    return enhanced_template

def add_execution_logic(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 2: Add Rule Execution Logic
    """
    when = rule.get('when', {})
    level = rule.get('level', '')

    # Build trigger condition string
    trigger_parts = []
    if level == 'MICRO':
        pa_id = when.get('pa_id', '')
        dim_id = when.get('dim_id', '')
        score_lt = when.get('score_lt', 1.65)
        trigger_parts.append(f"score < {score_lt}")
        trigger_parts.append(f"pa_id = '{pa_id}'")
        trigger_parts.append(f"dim_id = '{dim_id}'")
    elif level == 'MESO':
        cluster_id = when.get('cluster_id', '')
        score_band = when.get('score_band', '')
        if score_band:
            trigger_parts.append(f"score_band = '{score_band}'")
        if cluster_id:
            trigger_parts.append(f"cluster_id = '{cluster_id}'")
    elif level == 'MACRO':
        macro_band = when.get('macro_band', '')
        if macro_band:
            trigger_parts.append(f"macro_band = '{macro_band}'")

    trigger_condition = " AND ".join(trigger_parts) if trigger_parts else "true"

    return {
        "trigger_condition": trigger_condition,
        "blocking": False,
        "auto_apply": False,
        "requires_approval": True,
```

```
        "approval_roles": ["SecretarÃa de PlaneaciÃ³n", "SecretarÃa de Hacienda"]
    }

def enhance_indicator(indicator: dict[str, Any], rule_id: str, level: str) -> dict[str, Any]:
    """
    Feature 3: Make Indicators Measurable
    """
    enhanced = copy.deepcopy(indicator)

    # Add formula based on indicator type
    indicator.get('name', '')
    if 'proporciÃ³n' in indicator.get('unit', ''):
        enhanced['formula'] = 'COUNT(compliant_items) / COUNT(total_items)'
        enhanced['acceptable_range'] = [0.6, 1.0]
    elif 'porcentaje' in indicator.get('unit', ''):
        enhanced['formula'] = '(achieved / target) * 100'
        enhanced['acceptable_range'] = [60.0, 100.0]
    else:
        enhanced['formula'] = 'SUM(verified_artifacts)'
        enhanced['acceptable_range'] = [indicator.get('target', 1) * 0.7, indicator.get('target', 1)]

    # Add measurement metadata
    enhanced['baseline_measurement_date'] = '2024-01-01'
    enhanced['measurement_frequency'] = 'mensual'
    enhanced['data_source'] = 'Sistema de Seguimiento de Planes (SSP)'
    enhanced['data_source_query'] = f"SELECT COUNT(*) FROM indicators WHERE indicator_id = '{rule_id}-IND'"
    enhanced['responsible_measurement'] = 'Oficina de PlaneaciÃ³n Municipal'
    enhanced['escalation_if_below'] = enhanced['acceptable_range'][0]

    return enhanced

def enhance_horizon(horizon: dict[str, str], rule_id: str) -> dict[str, Any]:
    """
    Feature 4: Define Unambiguous Time Horizons
    """
    # Map T0, T1, T2, T3 to actual durations
    duration_map = {
        'T0': 0,
        'T1': 6,   # 6 months
        'T2': 12,  # 12 months
        'T3': 24   # 24 months
    }

    start = horizon.get('start', 'T0')
    end = horizon.get('end', 'T1')

    duration_months = duration_map.get(end, 6) - duration_map.get(start, 0)

    # Create milestones
    milestones = []
    if duration_months >= 6:
        milestones.append({
            "name": "Inicio de implementaciÃ³n",
            "offset_months": 1,
            "deliverables": ["Plan de trabajo aprobado"],
            "verification_required": True
```

```
        })
    if duration_months >= 12:
        milestones.append({
            "name": "RevisiÃ³n intermedia",
            "offset_months": duration_months // 2,
            "deliverables": ["Informe de avance"],
            "verification_required": True
        })
    milestones.append({
        "name": "Entrega final",
        "offset_months": duration_months,
        "deliverables": ["Todos los productos esperados"],
        "verification_required": True
    })

    return {
        "start": start,
        "end": end,
        "start_type": "plan_approval_date",
        "duration_months": duration_months,
        "milestones": milestones,
        "dependencies": [],
        "critical_path": duration_months <= 6
    }

def enhance_verification(verification: list[str], rule_id: str) -> list[dict[str, Any]]:
    """
    Feature 5: Make Verification Testable
    """
    enhanced_verifications = []

    for idx, artifact_text in enumerate(verification, 1):
        # Determine artifact type
        artifact_type = "DOCUMENT"
        if any(word in artifact_text.lower() for word in [’sistema’, ’repositorio’, ’registro’, ’base de datos’]):
            artifact_type = "SYSTEM_STATE"

        # Create structured verification
        ver_obj = {
            "id": f"VER-{rule_id}-{idx:03d}",
            "type": artifact_type,
            "artifact": artifact_text,
            "format": "PDF" if artifact_type == "DOCUMENT" else "DATABASE_QUERY",
            "required_sections": ["Objetivo", "Alcance", "Resultados"] if artifact_type == "DOCUMENT" else [],
            "approval_required": True,
            "approver": "SecretarÃa de PlaneaciÃ³n",
            "due_date": "T1",
            "automated_check": artifact_type == "SYSTEM_STATE"
        }

        # Add validation for system states
        if artifact_type == "SYSTEM_STATE":
            ver_obj["validation_query"] = f"SELECT COUNT(*) FROM artifacts WHERE artifact_id = ’{ver_obj[’id’]}’"
            ver_obj["pass_condition"] = "COUNT(*) >= 1"

        enhanced_verifications.append(ver_obj)
```

```

    return enhanced_verifications

def add_budget(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 6: Integrate Cost Tracking
    """
    level = rule.get(’level’, ’’)

    # Estimate costs based on level and complexity
    if level == ’MICRO’:
        base_cost = 45_000_000  # COP
    elif level == ’MESO’:
        base_cost = 150_000_000
    elif level == ’MACRO’:
        base_cost = 500_000_000
    else:
        base_cost = 50_000_000

    # Cost breakdown
    personal_cost = int(base_cost * 0.55)
    consultancy_cost = int(base_cost * 0.30)
    technology_cost = int(base_cost * 0.15)

    return {
        "estimated_cost_cop": base_cost,
        "cost_breakdown": {
            "personal": personal_cost,
            "consultancy": consultancy_cost,
            "technology": technology_cost
        },
        "funding_sources": [
            {
                "source": "SGP - Sistema General de Participaciones",
                "amount": int(base_cost * 0.60),
                "confirmed": False
            },
            {
                "source": "Recursos Propios",
                "amount": int(base_cost * 0.40),
                "confirmed": False
            }
        ],
        "fiscal_year": 2025
    }

def enhance_responsible(responsible: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 7: Map Authority for Accountability
    """
    enhanced = copy.deepcopy(responsible)

    # Add legal mandate
    entity = responsible.get(’entity’, ’’)
    if ’Mujer’ in entity:
        legal_mandate = "Ley 1257 de 2008 - Normas para la prevenciÃ³n de violencias contra la mujer"
```

```
    elif ’PlaneaciÃ³n’ in entity:
        legal_mandate = "Ley 152 de 1994 - Ley OrgÃ¡nica del Plan de Desarrollo"
    elif ’Hacienda’ in entity:
        legal_mandate = "Ley 819 de 2003 - Responsabilidad Fiscal"
    else:
        legal_mandate = "Estatuto OrgÃ¡nico Municipal"

    enhanced[’legal_mandate’] = legal_mandate

    # Add approval chain
    enhanced[’approval_chain’] = [
        {
            "level": 1,
            "role": "Director/Coordinador de Programa",
            "decision": "Aprueba plan de trabajo"
        },
        {
            "level": 2,
            "role": "Secretario/a de la entidad responsable",
            "decision": "Aprueba presupuesto y recursos"
        },
        {
            "level": 3,
            "role": "SecretarÃ-a de PlaneaciÃ³n",
            "decision": "Valida coherencia con PDM"
        },
        {
            "level": 4,
            "role": "Alcalde Municipal",
            "decision": "AprobaciÃ³n final (si aplica)"
        }
    ]

    # Add escalation path
    enhanced[’escalation_path’] = {
        "threshold_days_delay": 15,
        "escalate_to": "SecretarÃ-a de PlaneaciÃ³n",
        "final_escalation": "Despacho del Alcalde",
        "consequences": ["RevisiÃ³n presupuestal", "ReasignaciÃ³n de responsables"]
    }

    return enhanced

def enhance_rule(rule: dict[str, Any]) -> dict[str, Any]:
    """Enhance a single rule with all 7 features"""
    enhanced_rule = copy.deepcopy(rule)

    # 1. Template parameterization
    enhanced_rule[’template’] = enhance_template(rule)

    # 2. Execution logic
    enhanced_rule[’execution’] = add_execution_logic(rule)

    # 3. Measurable indicators
    if ’indicator’ in enhanced_rule[’template’]:
        enhanced_rule[’template’][’indicator’] = enhance_indicator(
```

```
                enhanced_rule['template']['indicator'],
                rule.get('rule_id', ''),
                rule.get('level', '')
            )

    # 4. Unambiguous time horizons
    if 'horizon' in enhanced_rule['template']:
        enhanced_rule['template']['horizon'] = enhance_horizon(
            enhanced_rule['template']['horizon'],
            rule.get('rule_id', '')
        )

    # 5. Testable verification
    if 'verification' in enhanced_rule['template']:
        enhanced_rule['template']['verification'] = enhance_verification(
            enhanced_rule['template']['verification'],
            rule.get('rule_id', '')
        )

    # 6. Budget tracking
    enhanced_rule['budget'] = add_budget(rule)

    # 7. Authority mapping
    if 'responsible' in enhanced_rule['template']:
        enhanced_rule['template']['responsible'] = enhance_responsible(
            enhanced_rule['template']['responsible']
        )

    return enhanced_rule

def main() -> None:
    """Main enhancement process"""
    # Delegate to factory for I/O operations
    from farfan_pipeline.analysis.factory import load_json, save_json

    # Load existing rules
    rules_path = Path('config/recommendation_rules.json')
    rules_data = load_json(rules_path)

    print(f"Loaded {len(rules_data['rules'])} rules from {rules_path}")

    # Enhance all rules
    enhanced_rules = []
    for i, rule in enumerate(rules_data['rules'], 1):
        try:
            enhanced = enhance_rule(rule)
            enhanced_rules.append(enhanced)
            if i % 10 == 0:
                print(f"Enhanced {i}/{len(rules_data['rules'])} rules...")
        except Exception as e:
            print(f"Error enhancing rule {rule.get('rule_id', 'UNKNOWN')}: {e}")
            enhanced_rules.append(rule)  # Keep original if enhancement fails

    # Create enhanced data structure
    enhanced_data = {
        'version': '2.0',  # Increment version
```

```
        'enhanced_features': [
            'template_parameterization',
            'execution_logic',
            'measurable_indicators',
            'unambiguous_time_horizons',
            'testable_verification',
            'cost_tracking',
            'authority_mapping'
        ],
        'rules': enhanced_rules
    }

    # Save enhanced rules
    output_path = Path('config/recommendation_rules_enhanced.json')
    save_json(enhanced_data, output_path)

    print(f"\nEnhanced {len(enhanced_rules)} rules saved to {output_path}")
    print(f"Original file preserved at {rules_path}")

    # Show sample enhanced rule
    if enhanced_rules:
        print("\n=== Sample Enhanced Rule ===")
        print(json.dumps(enhanced_rules[0], indent=2, ensure_ascii=False)[:2000])
        print("...")

# Note: Main entry point removed to maintain I/O boundary separation.
# For usage examples, see examples/ directory.



================================================================================
FILE: src/farfan_pipeline/analysis/factory.py
================================================================================

"""
Factory module â\200\224 canonical Dependency Injection (DI) and access control for F.A.R.F.A.N.

This module is the single authoritative boundary for:
- Canonical monolith access (CanonicalQuestionnaire)
- Signal registry construction and enrichment (QuestionnaireSignalRegistry v2.0)
- Method injection via MethodExecutor (with MethodRegistry + special instantiation rules)
- Orchestrator construction with strict DI
- CoreModuleFactory for I/O helpers (contracts, validation)

Design Principles (Factory Pattern + DI):
- Orchestrator and Executors never touch I/O nor load the monolith directly.
- Factory loads and validates the canonical questionnaire exactly once (singleton).
- Factory constructs signal registries and enriched packs centrally.
- Factory wires MethodExecutor with registries and special instantiation rules.
- Factory injects EnrichedSignalPack per policy area for BaseExecutor use.

Scope:
- Infrastructure layer only. No business logic.

SIN_CARRETA Compliance:
- All construction paths emit structured telemetry with timestamps and hashes.
```

```
- Determinism enforced via explicit validation of canonical questionnaire integrity.
- Contract assertions guard all factory outputs (no silent degradation).
- Auditability via immutable ProcessorBundle with provenance metadata.
- SeedRegistry singleton ensures deterministic stochastic operations.

Integration Points:
1. Orchestrator receives: method_executor, questionnaire, executor_config
2. BaseExecutor (30 classes) receives: enriched_signal_pack (via helper function)
3. MethodExecutor routes all method calls via ExtendedArgRouter
4. Special instantiation rules enable shared MunicipalOntology, dependency injection
"""

from __future__ import annotations

import hashlib
import json
import logging
import time
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Optional

from farfan_pipeline.core.orchestrator.core import MethodExecutor
from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
from farfan_pipeline.core.orchestrator.method_registry import (
    MethodRegistry,
    setup_default_instantiation_rules,
)
from farfan_pipeline.core.orchestrator.signal_registry import (
    QuestionnaireSignalRegistry,
    create_signal_registry,
)
from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
    EnrichedSignalPack,
    create_enriched_signal_pack,
)
from farfan_pipeline.core.orchestrator.questionnaire import (
    CanonicalQuestionnaire,
    load_questionnaire,
)
from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
from farfan_pipeline.core.orchestrator.class_registry import build_class_registry

# Optional: CoreModuleFactory for I/O helpers
try:
    from farfan_pipeline.core.orchestrator.core_module_factory import CoreModuleFactory
    CORE_MODULE_FACTORY_AVAILABLE = True
except ImportError:
    CoreModuleFactory = None  # type: ignore
    CORE_MODULE_FACTORY_AVAILABLE = False

# Optional: SeedRegistry for determinism
try:
    from farfan_pipeline.core.orchestrator.seed_registry import SeedRegistry
    SEED_REGISTRY_AVAILABLE = True
except ImportError:
```

```
    SeedRegistry = None  # type: ignore
    SEED_REGISTRY_AVAILABLE = False

logger = logging.getLogger(__name__)


# ============================================================================
# Exceptions
# ============================================================================


class FactoryError(Exception):
    """Base exception for factory construction failures."""
    pass


class QuestionnaireValidationError(FactoryError):
    """Raised when questionnaire validation fails."""
    pass


class RegistryConstructionError(FactoryError):
    """Raised when signal registry construction fails."""
    pass


class ExecutorConstructionError(FactoryError):
    """Raised when method executor construction fails."""
    pass


# ============================================================================
# Processor Bundle (typed DI container with provenance)
# ============================================================================


@dataclass(frozen=True)
class ProcessorBundle:
    """Aggregated orchestrator dependencies built by the Factory.

    Attributes:
        method_executor: Preconfigured MethodExecutor ready for routing.
        questionnaire: Immutable, validated CanonicalQuestionnaire.
        signal_registry: QuestionnaireSignalRegistry v2.0 with full metadata.
        executor_config: Canonical ExecutorConfig for executors.
        enriched_signal_packs: Dict of EnrichedSignalPack per policy area.
        core_module_factory: Optional CoreModuleFactory for I/O helpers.
        provenance: Construction metadata for audit trails.
    """

    method_executor: MethodExecutor
    questionnaire: CanonicalQuestionnaire
    signal_registry: QuestionnaireSignalRegistry
    executor_config: ExecutorConfig
    enriched_signal_packs: dict[str, EnrichedSignalPack]
    core_module_factory: Optional[Any] = None
```

```
    provenance: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self) -> None:
        """SIN_CARRETA Â§ Contract Enforcement: validate bundle integrity."""
        errors = []

        if self.method_executor is None:
            errors.append("method_executor must not be None")
        if self.questionnaire is None:
            errors.append("questionnaire must not be None")
        if self.signal_registry is None:
            errors.append("signal_registry must not be None")
        if self.executor_config is None:
            errors.append("executor_config must not be None")
        if self.enriched_signal_packs is None:
            errors.append("enriched_signal_packs must not be None")
        elif not isinstance(self.enriched_signal_packs, dict):
            errors.append("enriched_signal_packs must be dict[str, EnrichedSignalPack]")

        if not self.provenance.get("construction_timestamp_utc"):
            errors.append("provenance must include construction_timestamp_utc")
        if not self.provenance.get("canonical_sha256"):
            errors.append("provenance must include canonical_sha256")
        if self.provenance.get("signal_registry_version") != "2.0":
            errors.append("provenance must indicate signal_registry_version=2.0")

        if errors:
            raise FactoryError(f"ProcessorBundle validation failed: {’; ’.join(errors)}")

        logger.info(
            "processor_bundle_validated "
            "canonical_sha256=%s construction_ts=%s policy_areas=%d",
            self.provenance.get("canonical_sha256", "")[:16],
            self.provenance.get("construction_timestamp_utc"),
            len(self.enriched_signal_packs),
        )


# ============================================================================
# Core Factory Implementation
# ============================================================================


def build_processor_bundle(
    *,
    questionnaire_path: Optional[str] = None,
    executor_config: Optional[ExecutorConfig] = None,
    enable_intelligence_layer: bool = True,
    seed_for_determinism: Optional[int] = None,
    strict_validation: bool = True,
) -> ProcessorBundle:
    """Build complete processor bundle with all dependencies wired.

    This is the primary factory entry point for constructing all orchestrator
    dependencies in a single, validated operation.

```

```
    Args:
        questionnaire_path: Path to canonical questionnaire JSON. If None, uses default.
        executor_config: Custom executor configuration. If None, uses default.
        enable_intelligence_layer: Whether to build enriched signal packs (default: True).
        seed_for_determinism: Optional seed for reproducible stochastic operations.
        strict_validation: If True, fail on any validation error (default: True).

    Returns:
        ProcessorBundle: Immutable bundle with all dependencies wired and validated.

    Raises:
        QuestionnaireValidationError: If questionnaire validation fails.
        RegistryConstructionError: If signal registry construction fails.
        ExecutorConstructionError: If method executor construction fails.
        FactoryError: For other construction failures.
    """
    construction_start = time.time()
    timestamp_utc = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())

    logger.info("factory_build_start timestamp=%s strict=%s", timestamp_utc, strict_validation)

    try:
        # Step 1: Load and validate canonical questionnaire
        questionnaire = _load_and_validate_questionnaire(questionnaire_path, strict_validation)
        canonical_hash = _compute_questionnaire_hash(questionnaire)

        if not isinstance(questionnaire, CanonicalQuestionnaire):
            logger.error("Loaded questionnaire is not a CanonicalQuestionnaire instance: type=%s", type(questionnaire))
            num_questions = 0
        elif not hasattr(questionnaire, ’questions’) or not isinstance(questionnaire.questions, (list, tuple)):
            logger.error("CanonicalQuestionnaire missing ’questions’ attribute or it is not a list/tuple: %s", repr(questionnaire))
            num_questions = 0
        else:
            num_questions = len(questionnaire.questions)
        logger.info(
            "questionnaire_loaded questions=%d hash=%s",
            num_questions,
            canonical_hash[:16],
        )

        # Step 2: Build signal registry v2.0
        signal_registry = _build_signal_registry(questionnaire, strict_validation)

        if not hasattr(signal_registry, ’get_all_policy_areas’) or not callable(getattr(signal_registry, ’get_all_policy_areas’, None)):
            logger.error("signal_registry does not implement required method ’get_all_policy_areas’")
            raise AttributeError("signal_registry does not implement required method ’get_all_policy_areas’")
        logger.info(
            "signal_registry_built version=2.0 policy_areas=%d",
            len(signal_registry.get_all_policy_areas()),
        )

        # Step 3: Build enriched signal packs (intelligence layer)
        enriched_packs = _build_enriched_packs(
            signal_registry,
            questionnaire,
            enable_intelligence_layer,
```

```
            strict_validation
        )

        logger.info(
            "enriched_packs_built count=%d intelligence_layer=%s",
            len(enriched_packs),
            "enabled" if enable_intelligence_layer else "disabled",
        )

        # Step 4: Initialize seed registry for determinism
        _initialize_seed_registry(seed_for_determinism)

        # Step 5: Build method executor with full wiring
        method_executor = _build_method_executor(strict_validation)

        logger.info(
            "method_executor_built special_routes=%d",
            method_executor.arg_router.get_special_route_coverage() if hasattr(method_executor.arg_router, 'get_special_route_coverage') else 0,
        )

        # Step 6: Build or use provided executor config
        if executor_config is None:
            executor_config = ExecutorConfig.default()

        # Step 7: Build optional core module factory
        core_factory = _build_core_module_factory()

        # Step 8: Assemble provenance metadata
        construction_duration = time.time() - construction_start
        provenance = {
            "construction_timestamp_utc": timestamp_utc,
            "canonical_sha256": canonical_hash,
            "signal_registry_version": "2.0",
            "intelligence_layer_enabled": enable_intelligence_layer,
            "enriched_packs_count": len(enriched_packs),
            "construction_duration_seconds": round(construction_duration, 3),
            "seed_registry_initialized": SEED_REGISTRY_AVAILABLE and seed_for_determinism is not None,
            "core_module_factory_available": CORE_MODULE_FACTORY_AVAILABLE,
            "strict_validation": strict_validation,
        }

        # Step 9: Build and validate bundle
        bundle = ProcessorBundle(
            method_executor=method_executor,
            questionnaire=questionnaire,
            signal_registry=signal_registry,
            executor_config=executor_config,
            enriched_signal_packs=enriched_packs,
            core_module_factory=core_factory,
            provenance=provenance,
        )

        logger.info(
            "factory_build_complete duration=%.3fs hash=%s",
            construction_duration,
            canonical_hash[:16],
```

```
        )

        return bundle

    except Exception as e:
        logger.error("factory_build_failed error=%s", str(e), exc_info=True)
        raise FactoryError(f"Failed to build processor bundle: {e}") from e


# =============================================================================
# Internal Construction Functions
# =============================================================================


def _load_and_validate_questionnaire(
    path: Optional[str],
    strict: bool,
) -> CanonicalQuestionnaire:
    """Load and validate canonical questionnaire."""
    try:
        questionnaire_path = Path(path) if path is not None else None
        questionnaire = load_questionnaire(questionnaire_path)

        # Validate structure
        if not hasattr(questionnaire, 'questions'):
            if strict:
                raise QuestionnaireValidationError("Questionnaire missing 'questions' attribute")
            logger.warning("questionnaire_validation_warning missing_questions_attribute")

        questions = getattr(questionnaire, 'questions', [])
        if not questions:
            if strict:
                raise QuestionnaireValidationError("Questionnaire has no questions")
            logger.warning("questionnaire_validation_warning no_questions")

        return questionnaire

    except Exception as e:
        if strict:
            raise QuestionnaireValidationError(f"Failed to load questionnaire: {e}") from e
        logger.error("questionnaire_load_error continuing_with_degraded_state", exc_info=True)
        raise


def _build_signal_registry(
    questionnaire: CanonicalQuestionnaire,
    strict: bool,
) -> QuestionnaireSignalRegistry:
    """Build signal registry from questionnaire."""
    try:
        registry = create_signal_registry(questionnaire)

        # Validate registry
        if not hasattr(registry, 'get_all_policy_areas'):
            if strict:
                raise RegistryConstructionError("Registry missing required methods")
```

```
            logger.warning("registry_validation_warning missing_methods")

        return registry

    except Exception as e:
        if strict:
            raise RegistryConstructionError(f"Failed to build signal registry: {e}") from e
        logger.error("registry_construction_error", exc_info=True)
        raise


def _build_enriched_packs(
    signal_registry: QuestionnaireSignalRegistry,
    questionnaire: CanonicalQuestionnaire,
    enable: bool,
    strict: bool,
) -> dict[str, EnrichedSignalPack]:
    """Build enriched signal packs for all policy areas."""
    enriched_packs: dict[str, EnrichedSignalPack] = {}

    if not enable:
        logger.info("enriched_packs_disabled")
        return enriched_packs

    try:
        policy_areas = signal_registry.get_all_policy_areas() if hasattr(signal_registry, 'get_all_policy_areas') else []

        if not policy_areas:
            logger.warning("no_policy_areas_found registry_empty")
            return enriched_packs

        for policy_area_id in policy_areas:
            try:
                base_pack = signal_registry.get(policy_area_id) if hasattr(signal_registry, 'get') else None

                if base_pack is None:
                    logger.warning("base_pack_missing policy_area=%s", policy_area_id)
                    continue

                enriched_pack = create_enriched_signal_pack(
                    base_pack,
                    questionnaire,
                )
                enriched_packs[policy_area_id] = enriched_pack

            except Exception as e:
                msg = f"Failed to create enriched pack for {policy_area_id}: {e}"
                if strict:
                    raise RegistryConstructionError(msg) from e
                logger.error("enriched_pack_creation_failed policy_area=%s", policy_area_id, exc_info=True)

        return enriched_packs

    except Exception as e:
        if strict:
            raise RegistryConstructionError(f"Failed to build enriched packs: {e}") from e
```

```
        logger.error("enriched_packs_construction_error", exc_info=True)
        return enriched_packs


def _initialize_seed_registry(seed: Optional[int]) -> None:
    """Initialize seed registry if available."""
    if not SEED_REGISTRY_AVAILABLE:
        logger.debug("seed_registry_unavailable module_not_found")
        return

    if seed is None:
        logger.debug("seed_registry_not_initialized no_seed_provided")
        return

    try:
        SeedRegistry.initialize(master_seed=seed)
        logger.info("seed_registry_initialized master_seed=%d", seed)
    except Exception as e:
        logger.error("seed_registry_initialization_failed", exc_info=True)
        # Non-fatal, continue without determinism


def _build_method_executor(strict: bool) -> MethodExecutor:
    """Build method executor with full dependency wiring."""
    try:
        # Build method registry
        method_registry = MethodRegistry()
        setup_default_instantiation_rules(method_registry)

        # Build class registry
        class_registry = build_class_registry()

        # Build extended arg router
        arg_router = ExtendedArgRouter(class_registry)

        # Build method executor
        method_executor = MethodExecutor(
            method_registry=method_registry,
        )

        # Validate construction
        if not hasattr(method_executor, 'execute'):
            if strict:
                raise ExecutorConstructionError("MethodExecutor missing 'execute' method")
            logger.warning("method_executor_validation_warning missing_execute")

        return method_executor

    except Exception as e:
        if strict:
            raise ExecutorConstructionError(f"Failed to build method executor: {e}") from e
        logger.error("method_executor_construction_error", exc_info=True)
        raise


def _build_core_module_factory() -> Optional[Any]:
```

```
    """Build core module factory if available."""
    if not CORE_MODULE_FACTORY_AVAILABLE:
        logger.debug("core_module_factory_unavailable module_not_found")
        return None

    try:
        factory = CoreModuleFactory()
        logger.info("core_module_factory_built")
        return factory
    except Exception as e:
        logger.error("core_module_factory_construction_error", exc_info=True)
        return None


def _compute_questionnaire_hash(questionnaire: CanonicalQuestionnaire) -> str:
    """Compute deterministic SHA256 hash of questionnaire content."""
    try:
        # Try to get JSON representation if available
        if hasattr(questionnaire, 'to_dict'):
            content = json.dumps(questionnaire.to_dict(), sort_keys=True)
        elif hasattr(questionnaire, '__dict__'):
            content = json.dumps(questionnaire.__dict__, sort_keys=True, default=str)
        else:
            content = str(questionnaire)

        return hashlib.sha256(content.encode('utf-8')).hexdigest()

    except Exception as e:
        logger.warning("questionnaire_hash_computation_degraded error=%s", str(e))
        # Fallback to simple string hash
        return hashlib.sha256(str(questionnaire).encode('utf-8')).hexdigest()


# ============================================================================
# Convenience API
# ============================================================================


def build_processor(
    questionnaire_path: Optional[str] = None,
    seed: Optional[int] = None,
) -> ProcessorBundle:
    """
    Convenience wrapper for `build_processor_bundle` with sensible defaults.

    This function is intended for typical use cases where you want a fully configured
    processor with the intelligence layer enabled, strict validation, and optional
    reproducibility via a seed. It sets recommended defaults for most users.

    Use `build_processor_bundle` directly if you need advanced customization, such as
    disabling the intelligence layer, changing validation strictness, or other options.

    Args:
        questionnaire_path: Optional path to questionnaire JSON.
        seed: Optional seed for reproducibility.
    Returns:
```

```
        ProcessorBundle ready for use.
    """
    return build_processor_bundle(
        questionnaire_path=questionnaire_path,
        enable_intelligence_layer=True,
        seed_for_determinism=seed,
        strict_validation=True,
    )


def build_minimal_processor(
    questionnaire_path: Optional[str] = None,
    strict: bool = False,
) -> ProcessorBundle:
    """Build minimal processor bundle without intelligence layer.

    Useful for testing or when enriched signals are not needed.

    Args:
        questionnaire_path: Optional path to questionnaire JSON.
        strict: Whether to use strict validation (default: False for minimal).

    Returns:
        ProcessorBundle with basic dependencies only.
    """
    return build_processor_bundle(
        questionnaire_path=questionnaire_path,
        enable_intelligence_layer=False,
        strict_validation=strict,
    )


def get_enriched_pack_for_policy_area(
    bundle: ProcessorBundle,
    policy_area_id: str,
) -> Optional[EnrichedSignalPack]:
    """Helper to safely retrieve enriched signal pack from bundle.

    Args:
        bundle: Processor bundle.
        policy_area_id: Policy area identifier.

    Returns:
        EnrichedSignalPack if available, None otherwise.
    """
    return bundle.enriched_signal_packs.get(policy_area_id)


# ============================================================================
# Validation and Diagnostics
# ============================================================================


def validate_bundle(bundle: ProcessorBundle) -> dict[str, Any]:
    """Validate bundle integrity and return diagnostics.

```

```
    Args:
        bundle: ProcessorBundle to validate.

    Returns:
        Dictionary with validation results and diagnostics.
    """
    diagnostics = {
        "valid": True,
        "errors": [],
        "warnings": [],
        "components": {},
        "metrics": {},
    }

    # Validate method executor
    if bundle.method_executor is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("method_executor is None")
    else:
        diagnostics["components"]["method_executor"] = "present"
        if hasattr(bundle.method_executor, 'arg_router'):
            router = bundle.method_executor.arg_router
            if hasattr(router, 'get_special_route_coverage'):
                diagnostics["metrics"]["special_routes"] = router.get_special_route_coverage()

    # Validate questionnaire
    if bundle.questionnaire is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("questionnaire is None")
    else:
        diagnostics["components"]["questionnaire"] = "present"
        if hasattr(bundle.questionnaire, 'questions'):
            diagnostics["metrics"]["question_count"] = len(bundle.questionnaire.questions)

    # Validate signal registry
    if bundle.signal_registry is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("signal_registry is None")
    else:
        diagnostics["components"]["signal_registry"] = "present"
        if hasattr(bundle.signal_registry, 'get_all_policy_areas'):
            diagnostics["metrics"]["policy_areas"] = len(bundle.signal_registry.get_all_policy_areas())

    # Validate enriched packs
    pack_count = len(bundle.enriched_signal_packs)
    diagnostics["components"]["enriched_packs"] = pack_count
    diagnostics["metrics"]["enriched_pack_count"] = pack_count

    if pack_count == 0 and bundle.provenance.get("intelligence_layer_enabled"):
        diagnostics["warnings"].append("Intelligence layer enabled but no enriched packs available")

    # Validate provenance
    required_provenance = ["construction_timestamp_utc", "canonical_sha256", "signal_registry_version"]
    missing_provenance = [k for k in required_provenance if k not in bundle.provenance]
    if missing_provenance:
        diagnostics["valid"] = False
```

```
        diagnostics["errors"].append(f"Missing provenance: {missing_provenance}")

    # Check provenance metrics
    diagnostics["metrics"]["construction_duration"] = bundle.provenance.get("construction_duration_seconds", 0)
    diagnostics["metrics"]["canonical_hash"] = bundle.provenance.get("canonical_sha256", "")[:16]

    return diagnostics


def get_bundle_info(bundle: ProcessorBundle) -> dict[str, Any]:
    """Get human-readable information about bundle.

    Args:
        bundle: ProcessorBundle to inspect.

    Returns:
        Dictionary with bundle information.
    """
    return {
        "construction_time": bundle.provenance.get("construction_timestamp_utc"),
        "canonical_hash": bundle.provenance.get("canonical_sha256", "")[:16],
        "policy_areas": sorted(bundle.enriched_signal_packs.keys()),
        "policy_area_count": len(bundle.enriched_signal_packs),
        "intelligence_layer": bundle.provenance.get("intelligence_layer_enabled"),
        "core_factory": bundle.core_module_factory is not None,
        "construction_duration": bundle.provenance.get("construction_duration_seconds"),
        "strict_validation": bundle.provenance.get("strict_validation"),
        "seed_initialized": bundle.provenance.get("seed_registry_initialized"),
    }


# ============================================================================
# Singleton Cache (Optional)
# ============================================================================

_bundle_cache: Optional[ProcessorBundle] = None
_cache_key: Optional[str] = None


def get_or_build_bundle(
    questionnaire_path: Optional[str] = None,
    cache: bool = True,
    force_rebuild: bool = False,
) -> ProcessorBundle:
    """Get cached bundle or build new one.

    Args:
        questionnaire_path: Optional path to questionnaire JSON.
        cache: Whether to cache the bundle (default: True).
        force_rebuild: Force rebuild even if cached (default: False).

    Returns:
        ProcessorBundle (cached or newly built).
    """
    global _bundle_cache, _cache_key

```

```
    cache_key = questionnaire_path or "default"

    if not force_rebuild and cache and _bundle_cache is not None and _cache_key == cache_key:
        logger.debug("factory_cache_hit key=%s", cache_key)
        return _bundle_cache

    logger.debug("factory_cache_miss key=%s building_new force_rebuild=%s", cache_key, force_rebuild)
    bundle = build_processor(questionnaire_path=questionnaire_path)

    if cache:
        _bundle_cache = bundle
        _cache_key = cache_key
        logger.debug("factory_cache_updated key=%s", cache_key)

    return bundle


def clear_bundle_cache() -> None:
    """Clear singleton bundle cache."""
    global _bundle_cache, _cache_key
    _bundle_cache = None
    _cache_key = None
    logger.debug("factory_cache_cleared")


def get_cache_info() -> dict[str, Any]:
    """Get information about current cache state."""
    return {
        "cached": _bundle_cache is not None,
        "cache_key": _cache_key,
        "bundle_hash": _bundle_cache.provenance.get("canonical_sha256", "")[:16] if _bundle_cache else None,
    }



================================================================================
FILE: src/farfan_pipeline/analysis/financiero_viabilidad_tablas.py
================================================================================

"""
MUNICIPAL DEVELOPMENT PLAN ANALYZER - PDET COLOMBIA
===================================================
VersiÃ³n: 5.0 - Causal Inference Edition (2025)
EspecializaciÃ³n: Planes de Desarrollo Municipal con AnÃ¡lisis Causal Bayesiano
Arquitectura: ExtracciÃ³n Avanzada + Inferencia Causal + DAG Learning + Counterfactuals

NUEVA CAPACIDAD - INFERENCIA CAUSAL:
â\234\223 IdentificaciÃ³n automÃ¡tica de mecanismos causales en PDM
â\234\223 ConstrucciÃ³n de DAGs (Directed Acyclic Graphs) para pilares PDET
â\234\223 EstimaciÃ³n bayesiana de efectos causales directos e indirectos
â\234\223 AnÃ¡lisis contrafactual de intervenciones
â\234\223 CuantificaciÃ³n de heterogeneidad causal por contexto territorial
â\234\223 DetecciÃ³n de confounders y mediadores
â\234\223 AnÃ¡lisis de sensibilidad para supuestos de identificaciÃ³n

COMPLIANCE:
```

```
â\234\223 Python 3.10+ con type hints completos
â\234\223 Sin placeholders - 100% implementado y probado
â\234\223 IntegraciÃ³n completa con pipeline existente
â\234\223 Calibrado para estructura de PDM colombianos
"""
from __future__ import annotations

import asyncio
import logging
import re
from dataclasses import dataclass, field
from datetime import datetime
from decimal import Decimal
from pathlib import Path
from typing import Any, Literal

# === EXTRACCIÃ\223N AVANZADA DE PDF Y TABLAS ===
import camelot

# === NETWORKING Y GRAFOS CAUSALES ===
import networkx as nx

# === CORE SCIENTIFIC COMPUTING ===
import numpy as np
import pandas as pd

# === ESTADÃ\215STICA BAYESIANA Y CAUSAL INFERENCE ===
import pymc as pm
import spacy
import tabula
import torch
from scipy import stats

# === NLP Y TRANSFORMERS ===
# Check dependency lockdown before importing transformers
from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
from sentence_transformers import SentenceTransformer, util
from sklearn.cluster import DBSCAN, AgglomerativeClustering

# === MACHINE LEARNING Y SCORING ===
from sklearn.feature_extraction.text import TfidfVectorizer
from transformers import pipeline
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

_lockdown = get_dependency_lockdown()

# ============================================================================
# LOGGING CONFIGURATION
# ============================================================================
logger = logging.getLogger(__name__)

# ============================================================================
# CONFIGURACIÃ\223N ESPECÃ\215FICA PARA COLOMBIA Y PDET
# ============================================================================

```

```
class ColombianMunicipalContext:
    """Contexto especÃ-fico del marco normativo colombiano para PDM"""

    OFFICIAL_SYSTEMS: dict[str, str] = {
        'SISBEN': r'SISB[EÃ\211]N\s*(?:I{1,4}|IV)?',
        'SGP': r'Sistema\s+General\s+de\s+Participaciones|SGP',
        'SGR': r'Sistema\s+General\s+de\s+Regal[Ã-i]as|SGR',
        'FUT': r'Formulario\s+[Ã\232U]nico\s+Territorial|FUT',
        'MFMP': r'Marco\s+Fiscal\s+(?:de\s+)?Mediano\s+Plazo|MFMP',
        'CONPES': r'CONPES\s*\d{3,4}',
        'DANE': r'(?:DANE|C[Ã³o]digo\s+DANE)\s*[:\-]?\s*(\d{5,8})',
        'MGA': r'Metodolog[Ã-i]a\s+General\s+Ajustada|MGA',
        'POAI': r'Plan\s+Operativo\s+Anual\s+de\s+Inversiones|POAI'
    }

    TERRITORIAL_CATEGORIES: dict[int, dict[str, Any]] = {
        1: {'name': 'Especial', 'min_pop': 500_001, 'min_income_smmlv': 400_000},
        2: {'name': 'Primera', 'min_pop': 100_001, 'min_income_smmlv': 100_000},
        3: {'name': 'Segunda', 'min_pop': 50_001, 'min_income_smmlv': 50_000},
        4: {'name': 'Tercera', 'min_pop': 30_001, 'min_income_smmlv': 30_000},
        5: {'name': 'Cuarta', 'min_pop': 20_001, 'min_income_smmlv': 25_000},
        6: {'name': 'Quinta', 'min_pop': 10_001, 'min_income_smmlv': 15_000},
        7: {'name': 'Sexta', 'min_pop': 0, 'min_income_smmlv': 0}
    }

    DNP_DIMENSIONS: list[str] = [
        'DimensiÃ³n EconÃ³mica',
        'DimensiÃ³n Social',
        'DimensiÃ³n Ambiental',
        'DimensiÃ³n Institucional',
        'DimensiÃ³n Territorial'
    ]

    PDET_PILLARS: list[str] = [
        'Ordenamiento social de la propiedad rural',
        'Infraestructura y adecuaciÃ³n de tierras',
        'Salud rural',
        'EducaciÃ³n rural y primera infancia',
        'Vivienda, agua potable y saneamiento bÃ¡sico',
        'ReactivaciÃ³n econÃ³mica y producciÃ³n agropecuaria',
        'Sistema para la garantÃ-a progresiva del derecho a la alimentaciÃ³n',
        'ReconciliaciÃ³n, convivencia y paz'
    ]

    PDET_THEORY_OF_CHANGE: dict[str, dict[str, Any]] = {
        'Ordenamiento social de la propiedad rural': {
            'outcomes': ['seguridad_juridica', 'reduccion_conflictos_tierra'],
            'mediators': ['formalizacion', 'acceso_justicia'],
            'lag_years': 3
        },
        'Infraestructura y adecuaciÃ³n de tierras': {
            'outcomes': ['conectividad', 'productividad_agricola'],
            'mediators': ['vias_terciarias', 'distritos_riego'],
            'lag_years': 2
        },
        'Salud rural': {
```

```
            'outcomes': ['mortalidad_infantil', 'esperanza_vida'],
            'mediators': ['cobertura_salud', 'infraestructura_salud'],
            'lag_years': 4
        },
        'EducaciÃ³n rural y primera infancia': {
            'outcomes': ['cobertura_educativa', 'calidad_educativa'],
            'mediators': ['infraestructura_escolar', 'docentes_calificados'],
            'lag_years': 5
        },
        'Vivienda, agua potable y saneamiento bÃ¡sico': {
            'outcomes': ['deficit_habitacional', 'enfermedades_hidricas'],
            'mediators': ['cobertura_acueducto', 'viviendas_dignas'],
            'lag_years': 3
        },
        'ReactivaciÃ³n econÃ³mica y producciÃ³n agropecuaria': {
            'outcomes': ['ingreso_rural', 'empleo_rural'],
            'mediators': ['credito_rural', 'asistencia_tecnica'],
            'lag_years': 2
        },
        'Sistema para la garantÃa progresiva del derecho a la alimentaciÃ³n': {
            'outcomes': ['seguridad_alimentaria', 'nutricion_infantil'],
            'mediators': ['produccion_local', 'acceso_alimentos'],
            'lag_years': 2
        },
        'ReconciliaciÃ³n, convivencia y paz': {
            'outcomes': ['cohesion_social', 'confianza_institucional'],
            'mediators': ['espacios_participacion', 'justicia_transicional'],
            'lag_years': 6
        }
    }

    INDICATOR_STRUCTURE: dict[str, list[str]] = {
        'resultado': ['lÃnea_base', 'meta', 'aÃ±o_base', 'aÃ±o_meta', 'fuente', 'responsable'],
        'producto': ['indicador', 'fÃ³rmula', 'unidad_medida', 'lÃnea_base', 'meta', 'periodicidad'],
        'gestiÃ³n': ['eficacia', 'eficiencia', 'economÃa', 'costo_beneficio']
    }

# ============================================================================
# ESTRUCTURAS DE DATOS
# ============================================================================

@dataclass
class CausalNode:
    """Nodo en el grafo causal"""
    name: str
    node_type: Literal['pilar', 'outcome', 'mediator', 'confounder']
    embedding: np.ndarray | None = None
    associated_budget: Decimal | None = None
    temporal_lag: int = 0
    evidence_strength: float = 0.0

@dataclass
class CausalEdge:
    """Arista causal entre nodos"""
    source: str
    target: str
```

```
    edge_type: Literal['direct', 'mediated', 'confounded']
    effect_size_posterior: tuple[float, float, float] | None = None
    mechanism: str = ""
    evidence_quotes: list[str] = field(default_factory=list)
    probability: float = 0.0

@dataclass
class CausalDAG:
    """Grafo AcÃ-clico Dirigido completo"""
    nodes: dict[str, CausalNode]
    edges: list[CausalEdge]
    adjacency_matrix: np.ndarray
    graph: nx.DiGraph

@dataclass
class CausalEffect:
    """Efecto causal estimado"""
    treatment: str
    outcome: str
    effect_type: Literal['ATE', 'ATT', 'direct', 'indirect', 'total']
    point_estimate: float
    posterior_mean: float
    credible_interval_95: tuple[float, float]
    probability_positive: float
    probability_significant: float
    mediating_paths: list[list[str]] = field(default_factory=list)
    confounders_adjusted: list[str] = field(default_factory=list)

@dataclass
class CounterfactualScenario:
    """Escenario contrafactual"""
    intervention: dict[str, float]
    predicted_outcomes: dict[str, tuple[float, float, float]]
    probability_improvement: dict[str, float]
    narrative: str

@dataclass
class ExtractedTable:
    df: pd.DataFrame
    page_number: int
    table_type: str | None
    extraction_method: Literal['camelot_lattice', 'camelot_stream', 'tabula', 'pdfplumber']
    confidence_score: float
    is_fragmented: bool = False
    continuation_of: int | None = None

@dataclass
class FinancialIndicator:
    source_text: str
    amount: Decimal
    currency: str
    fiscal_year: int | None
    funding_source: str
    budget_category: str
    execution_percentage: float | None
    confidence_interval: tuple[float, float]
```

```
    risk_level: float

@dataclass
class ResponsibleEntity:
    name: str
    entity_type: Literal['secretarÃa', 'oficina', 'direcciÃ³n', 'alcaldÃa', 'externo']
    specificity_score: float
    mentioned_count: int
    associated_programs: list[str]
    associated_indicators: list[str]
    budget_allocated: Decimal | None

@dataclass
class QualityScore:
    overall_score: float
    financial_feasibility: float
    indicator_quality: float
    responsibility_clarity: float
    temporal_consistency: float
    pdet_alignment: float
    causal_coherence: float
    confidence_interval: tuple[float, float]
    evidence: dict[str, Any]

# ============================================================================
# MOTOR PRINCIPAL
# ============================================================================

class PDETMunicipalPlanAnalyzer:
    """Analizador de vanguardia para Planes de Desarrollo Municipal PDET"""

    def __init__(self, use_gpu: bool = True, language: str = 'es', confidence_threshold: float = 0.7) -> None:
        self.device = 'cuda' if use_gpu and torch.cuda.is_available() else 'cpu'
        self.confidence_threshold = confidence_threshold
        self.context = ColombianMunicipalContext()

        print("ð\237\224§ Inicializando modelos de vanguardia...")

        self.semantic_model = SentenceTransformer(
            'sentence-transformers/paraphrase-multilingual-mpnet-base-v2',
            device=self.device
        )

        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import load_spacy_model

        try:
            self.nlp = load_spacy_model("es_dep_news_trf")
        except OSError:
            raise RuntimeError(
                "Modelo SpaCy 'es_dep_news_trf' no instalado. "
                "Ejecuta: python -m spacy download es_dep_news_trf"
            )

        self.entity_classifier = pipeline(
            "token-classification",
```

```
            model="mrm8488/bert-spanish-cased-finetuned-ner",
            device=0 if use_gpu else -1,
            aggregation_strategy="simple"
        )

        self.tfidf = TfidfVectorizer(
            max_features=1000,
            ngram_range=(1, 3),
            min_df=2,
            stop_words=self._get_spanish_stopwords()
        )

        self.pdet_embeddings = {
            pillar: self.semantic_model.encode(pillar, convert_to_tensor=False)
            for pillar in self.context.PDET_PILLARS
        }

        print("â\234\205 Modelos inicializados correctamente\n")

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords")
    def _get_spanish_stopwords(self) -> list[str]:
        base_stopwords = spacy.lang.es.stop_words.STOP_WORDS
        gov_stopwords = {
            'artÃ-culo', 'decreto', 'mediante', 'conforme', 'respecto',
            'acuerdo', 'resoluciÃ³n', 'ordenanza', 'literal', 'numeral'
        }
        return list(base_stopwords | gov_stopwords)

    # ========================================================================
    # EXTRACCIÃ\223N DE TABLAS
    # ========================================================================

    async def extract_tables(self, pdf_path: str) -> list[ExtractedTable]:
        print("ð\237\223\212 Iniciando extracciÃ³n avanzada de tablas...")
        all_tables: list[ExtractedTable] = []
        pdf_path_str = str(pdf_path)

        # Camelot Lattice
        try:
            lattice_tables = camelot.read_pdf(
                pdf_path_str, pages='all', flavor='lattice',
                line_scale=40, joint_tol=10, edge_tol=50
            )
            for idx, table in enumerate(lattice_tables):
                if table.parsing_report['accuracy'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords", "auto_param_L342_54", 0.7):
                    all_tables.append(ExtractedTable(
                        df=self._clean_dataframe(table.df),
                        page_number=table.page,
                        table_type=None,
                        extraction_method='camelot_lattice',
                        confidence_score=table.parsing_report['accuracy']
                    ))
        except Exception as e:
            print(f" â\232 ï¸\217 Camelot Lattice: {str(e)[:50]}")
```

```
        # Camelot Stream
        try:
            stream_tables = camelot.read_pdf(
                pdf_path_str, pages='all', flavor='stream',
                edge_tol=500, row_tol=15, column_tol=10
            )
            for idx, table in enumerate(stream_tables):
                if table.parsing_report['accuracy'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords", "auto_param_L360_54", 0.6):
                    all_tables.append(ExtractedTable(
                        df=self._clean_dataframe(table.df),
                        page_number=table.page,
                        table_type=None,
                        extraction_method='camelot_stream',
                        confidence_score=table.parsing_report['accuracy']
                    ))
        except Exception as e:
            print(f" â\232 ï¸\217 Camelot Stream: {str(e)[:50]}")

        # Tabula
        try:
            tabula_tables = tabula.read_pdf(
                pdf_path_str, pages='all', multiple_tables=True,
                stream=True, guess=True, silent=True
            )
            for idx, df in enumerate(tabula_tables):
                if not df.empty and len(df) > 2:
                    all_tables.append(ExtractedTable(
                        df=self._clean_dataframe(df),
                        page_number=idx + 1,
                        table_type=None,
                        extraction_method='tabula',
                        confidence_score = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords", "confidence_score", 0.6) # Refactored
                    ))
        except Exception as e:
            print(f" â\232 ï¸\217 Tabula: {str(e)[:50]}")

        unique_tables = self._deduplicate_tables(all_tables)
        print(f"â\234\205 {len(unique_tables)} tablas Ãºnicas extraÃdas\n")

        reconstructed = await self._reconstruct_fragmented_tables(unique_tables)
        print(f"ð\237\224\227 {len(reconstructed)} tablas despuÃ©s de reconstituciÃ³n\n")

        classified = self._classify_tables(reconstructed)
        return classified

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._clean_dataframe")
    def _clean_dataframe(self, df: pd.DataFrame) -> pd.DataFrame:
        if df.empty:
            return df
        df = df.dropna(how='all').reset_index(drop=True)
        df = df.dropna(axis=1, how='all')

        if len(df) > 0:
            first_row = df.iloc[0].astype(str)
```

```
            if self._is_likely_header(first_row):
                df.columns = first_row.values
                df = df.iloc[1:].reset_index(drop=True)

        for col in df.columns:
            df[col] = df[col].astype(str).str.strip()
            df[col] = df[col].replace(['', 'nan', 'None'], np.nan)

        return df

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._is_likely_header")
    def _is_likely_header(self, row: pd.Series, **kwargs) -> bool:
        """
        Determine if a DataFrame row is likely a header row based on linguistic analysis.

        Args:
            row: pandas Series representing a row from a DataFrame
            **kwargs: Accepts additional keyword arguments for backward compatibility.
                     These are ignored (e.g., pdf_path if mistakenly passed).

        Returns:
            Boolean indicating whether the row appears to be a header

        Note:
            This function only requires 'row' parameter. Any additional kwargs
            (like 'pdf_path') are silently ignored to maintain interface stability.
        """
        # Log warning if unexpected kwargs are passed
        if kwargs:
            logger.warning(
                f"_is_likely_header received unexpected keyword arguments: {list(kwargs.keys())}. "
                "These will be ignored. Expected signature: _is_likely_header(self, row: pd.Series)"
            )

        text = ' '.join(row.astype(str))
        doc = self.nlp(text)
        pos_counts = pd.Series([token.pos_ for token in doc]).value_counts()
        noun_ratio = pos_counts.get('NOUN', 0) / max(len(doc), 1)
        verb_ratio = pos_counts.get('VERB', 0) / max(len(doc), 1)
        return noun_ratio > verb_ratio and len(text) < 200

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables")
    def _deduplicate_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
        if len(tables) <= 1:
            return tables

        embeddings = []
        for table in tables:
            table_text = table.df.to_string()[:1000]
            emb = self.semantic_model.encode(table_text, convert_to_tensor=True)
            embeddings.append(emb)

        similarities = util.cos_sim(torch.stack(embeddings), torch.stack(embeddings))

        to_keep = []
        seen = set()
```

```
        for i, table in enumerate(tables):
            if i in seen:
                continue
            duplicates = (similarities[i] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables", "auto_param_L466_44", 0.85)).nonzero(as_tuple=True)[0].tolist()
            best_idx = max(duplicates, key=lambda idx: tables[idx].confidence_score)
            to_keep.append(tables[best_idx])
            seen.update(duplicates)

        return to_keep

    async def _reconstruct_fragmented_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
        if len(tables) < 2:
            return tables

        features = []
        for table in tables:
            col_structure = '|'.join(sorted(str(c)[:20] for c in table.df.columns))
            dtypes = '|'.join(sorted(str(dt) for dt in table.df.dtypes))
            content = table.df.to_string()[:500]
            combined = f"{col_structure} {dtypes} {content}"
            features.append(combined)

        embeddings = self.semantic_model.encode(features, convert_to_tensor=False)
        clustering = DBSCAN(eps=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables", "auto_param_L486_32", 0.3), min_samples=2, metric='cosine').fit(embeddings)

        reconstructed = []
        processed = set()
        for cluster_id in set(clustering.labels_):
            if cluster_id == -1:
                continue
            cluster_indices = np.where(clustering.labels_ == cluster_id)[0]
            if len(cluster_indices) > 1:
                sorted_indices = sorted(cluster_indices, key=lambda i: tables[i].page_number)
                dfs_to_concat = [tables[i].df for i in sorted_indices]
                merged_df = pd.concat(dfs_to_concat, ignore_index=True)
                main_table = tables[sorted_indices[0]]
                reconstructed.append(ExtractedTable(
                    df=merged_df,
                    page_number=main_table.page_number,
                    table_type=main_table.table_type,
                    extraction_method=main_table.extraction_method,
                    confidence_score=np.mean([tables[i].confidence_score for i in sorted_indices]),
                    is_fragmented=True,
                    continuation_of=None
                ))
                processed.update(sorted_indices)

        for i, table in enumerate(tables):
            if i not in processed:
                reconstructed.append(table)

        return reconstructed

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._classify_tables")
```

```
    def _classify_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
        classification_patterns = {
            'presupuesto': ['presupuesto', 'recursos', 'millones', 'sgp', 'sgr', 'fuente', 'financiaciÃ³n'],
            'indicadores': ['indicador', 'lÃnea base', 'meta', 'fÃ³rmula', 'unidad de medida', 'periodicidad'],
            'cronograma': ['cronograma', 'actividad', 'mes', 'trimestre', 'aÃ±o', 'fecha'],
            'responsables': ['responsable', 'secretarÃa', 'direcciÃ³n', 'oficina', 'ejecutor'],
            'diagnostico': ['diagnÃ³stico', 'problema', 'causa', 'efecto', 'situaciÃ³n actual'],
            'pdet': ['pdet', 'iniciativa', 'pilar', 'patr', 'transformaciÃ³n regional']
        }

        for table in tables:
            table_text = table.df.to_string().lower()
            scores = {}
            for table_type, keywords in classification_patterns.items():
                score = sum(1 for kw in keywords if kw in table_text)
                scores[table_type] = score

            if max(scores.values()) > 0:
                table.table_type = max(scores, key=scores.get)

        return tables

    # ============================================================================
    # ANÃ\201LISIS FINANCIERO
    # ============================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.analyze_financial_feasibility")
    def analyze_financial_feasibility(self, tables: list[ExtractedTable], text: str) -> dict[str, Any]:
        print("ð\237\222° Analizando feasibility financiero...")

        financial_indicators = self._extract_financial_amounts(text, tables)
        funding_sources = self._analyze_funding_sources(financial_indicators, tables)
        sustainability = self._assess_financial_sustainability(financial_indicators, funding_sources)
        risk_assessment = self._bayesian_risk_inference(financial_indicators, funding_sources, sustainability)

        return {
            'total_budget': sum(ind.amount for ind in financial_indicators),
            'financial_indicators': [self._indicator_to_dict(ind) for ind in financial_indicators],
            'funding_sources': funding_sources,
            'sustainability_score': sustainability,
            'risk_assessment': risk_assessment,
            'confidence': risk_assessment['confidence_interval']
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts")
    def _extract_financial_amounts(self, text: str, tables: list[ExtractedTable]) -> list[FinancialIndicator]:
        patterns = [
            r'\$?\s*(\d{1,3}(?:[.,]\d{3})*(?:[.,]\d{1,2})?)\s*millones?',
            r'\$?\s*(\d{1,3}(?:[.,]\d{3})*(?:[.,]\d{1,2})?)\s*(?:mil\s+)?millones?',
            r'\$\s*(\d{1,3}(?:[.,]\d{3})*(?:[.,]\d{1,2})?)',
            r'(\d{1,6})\s*SMMLV'
        ]

        indicators = []
        for pattern in patterns:
            for match in re.finditer(pattern, text, re.IGNORECASE):
```

```
                amount_str = match.group(1).replace('.', '').replace(',', '.')
                try:
                    amount = Decimal(amount_str)
                    if 'millon' in match.group(0).lower():
                        amount *= Decimal('1000000')

                    context_start = max(0, match.start() - 200)
                    context_end = min(len(text), match.end() + 200)
                    context = text[context_start:context_end]

                    funding_source = self._identify_funding_source(context)
                    year_match = re.search(r'20\d{2}', context)
                    fiscal_year = int(year_match.group()) if year_match else None

                    indicators.append(FinancialIndicator(
                        source_text=match.group(0),
                        amount=amount,
                        currency='COP',
                        fiscal_year=fiscal_year,
                        funding_source=funding_source,
                        budget_category='',
                        execution_percentage=None,
                        confidence_interval=(ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts", "auto_param_L595_45", 0.0), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts", "auto_param_L595_50", 0.0)),
                        risk_level = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts", "risk_level", 0.0) # Refactored
                    ))
                except (ValueError, Exception):
                    continue

        budget_tables = [t for t in tables if t.table_type == 'presupuesto']
        for table in budget_tables:
            table_indicators = self._extract_from_budget_table(table.df)
            indicators.extend(table_indicators)

        print(f" â\234\223 {len(indicators)} indicadores financieros extraÃdos")
        return indicators

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_funding_source")
    def _identify_funding_source(self, context: str) -> str:
        sources = {
            'SGP': ['sgp', 'sistema general de participaciones'],
            'SGR': ['sgr', 'regalÃas', 'sistema general de regalÃas'],
            'Recursos Propios': ['recursos propios', 'propios', 'ingresos corrientes'],
            'CofinanciaciÃ³n': ['cofinanciaciÃ³n', 'cofinanciado'],
            'CrÃ©dito': ['crÃ©dito', 'prÃ©stamo', 'endeudamiento'],
            'CooperaciÃ³n': ['cooperaciÃ³n internacional', 'donaciÃ³n'],
            'PDET': ['pdet', 'paz', 'transformaciÃ³n regional']
        }

        context_lower = context.lower()
        for source_name, keywords in sources.items():
            if any(kw in context_lower for kw in keywords):
                return source_name
        return 'No especificada'
```

```

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table")
    def _extract_from_budget_table(self, df: pd.DataFrame) -> list[FinancialIndicator]:
        indicators = []
        amount_cols = [col for col in df.columns if any(
            kw in str(col).lower() for kw in ['monto', 'valor', 'presupuesto', 'recursos']
        )]
        source_cols = [col for col in df.columns if any(
            kw in str(col).lower() for kw in ['fuente', 'financiaciÃ³n', 'origen']
        )]

        if not amount_cols:
            return indicators

        amount_col = amount_cols[0]
        source_col = source_cols[0] if source_cols else None

        for _, row in df.iterrows():
            try:
                amount_str = str(row[amount_col])
                amount_str = re.sub(r'[^\d.,]', '', amount_str)
                if not amount_str:
                    continue
                amount = Decimal(amount_str.replace('.', '').replace(',', '.'))
                funding_source = str(row[source_col]) if source_col else 'No especificada'

                indicators.append(FinancialIndicator(
                    source_text=f"Tabla: {amount_str}",
                    amount=amount,
                    currency='COP',
                    fiscal_year=None,
                    funding_source=funding_source,
                    budget_category='',
                    execution_percentage=None,
                    confidence_interval=(ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table", "auto_param_L660_41", 0.0), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table", "auto_param_L660_46", 0.0)),
                    risk_level = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table", "risk_level", 0.0) # Refactored
                ))
            except Exception:
                continue

        return indicators

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._analyze_funding_sources")
    def _analyze_funding_sources(self, indicators: list[FinancialIndicator], tables: list[ExtractedTable]) -> dict[
        str, Any]:
        source_distribution = {}
        for ind in indicators:
            source = ind.funding_source
            source_distribution[source] = source_distribution.get(source, Decimal(0)) + ind.amount

        total = sum(source_distribution.values())
        if total == 0:
            return {'distribution': {}, 'diversity_index': ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyze
```

```
r._analyze_funding_sources", "auto_param_L678_59", 0.0)}

        proportions = [float(amount / total) for amount in source_distribution.values()]
        diversity = -sum(p * np.log(p) if p > 0 else 0 for p in proportions)

        return {
            'distribution': {k: float(v) for k, v in source_distribution.items()},
            'diversity_index': float(diversity),
            'max_diversity': np.log(len(source_distribution)),
            'dependency_risk': ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._analyze_funding_sources", "auto_param_L687_31", 1.0) - (diversity / np.log(max(len(source_distribution), 2)))
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability")
    def _assess_financial_sustainability(self, indicators: list[FinancialIndicator],
                                         funding_sources: dict[str, Any]) -> float:
        if not indicators:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L694_19", 0.0)

        diversity_score = min(funding_sources.get('diversity_index', 0) / funding_sources.get('max_diversity', 1), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L696_115", 1.0))

        distribution = funding_sources.get('distribution', {})
        total = sum(distribution.values())
        own_resources = distribution.get('Recursos Propios', 0) / total if total > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L700_90", 0.0)
        pdet_dependency = distribution.get('PDET', 0) / total if total > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L701_80", 0.0)
        pdet_risk = min(pdet_dependency * 2, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L702_45", 1.0))

        sustainability = (diversity_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L704_44", 0.3) + own_resources * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L704_66", 0.4) + (1 - pdet_risk) * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L704_90", 0.3))
        return float(sustainability)

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference")
    def _bayesian_risk_inference(self, indicators: list[FinancialIndicator], funding_sources: dict[str, Any],
                                 sustainability: float) -> dict[str, Any]:
        print(" ð\237\216² Ejecutando inferencia bayesiana...")

        observed_data = {
            'n_indicators': len(indicators),
            'diversity': funding_sources.get('diversity_index', 0),
            'sustainability': sustainability,
            'dependency': funding_sources.get('dependency_risk', ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L716_65", 0.5))
        }

        with pm.Model():
            base_risk = pm.Beta('base_risk', alpha=2, beta=5)
            diversity_effect = pm.Normal('diversity_effect', mu=-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L721_65", 0.3), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L721_76", 0.1))
```

```
            sustainability_effect = pm.Normal('sustainability_effect', mu=-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L722_75", 0.4), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L722_86", 0.1))
            dependency_effect = pm.Normal('dependency_effect', mu=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L723_66", 0.5), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L723_77", 0.15))

            pm.Deterministic(
                'risk',
                pm.math.sigmoid(
                    pm.math.log(base_risk / (1 - base_risk)) +
                    diversity_effect * observed_data['diversity'] +
                    sustainability_effect * observed_data['sustainability'] +
                    dependency_effect * observed_data['dependency']
                )
            )

            trace = pm.sample(2000, tune=1000, cores=1, return_inferencedata=True, progressbar=False)

        risk_samples = trace.posterior['risk'].values.flatten()
        risk_mean = float(np.mean(risk_samples))
        risk_ci = tuple(float(x) for x in np.percentile(risk_samples, [2.5, 97.5]))

        print(f" â\234\223 Riesgo estimado: {risk_mean:.3f} CI95%: {risk_ci}")

        return {
            'risk_score': risk_mean,
            'confidence_interval': risk_ci,
            'interpretation': self._interpret_risk(risk_mean),
            'posterior_samples': risk_samples.tolist()
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_risk")
    def _interpret_risk(self, risk: float) -> str:
        if risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_risk", "auto_param_L752_18", 0.2):
            return "Riesgo bajo - Plan financieramente robusto"
        elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_risk", "auto_param_L754_20", 0.4):
            return "Riesgo moderado-bajo - Sostenibilidad probable"
        elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_risk", "auto_param_L756_20", 0.6):
            return "Riesgo moderado - Requiere monitoreo"
        elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_risk", "auto_param_L758_20", 0.8):
            return "Riesgo alto - Vulnerabilidades significativas"
        else:
            return "Riesgo crÃtico - Inviabilidad financiera probable"

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._indicator_to_dict")
    def _indicator_to_dict(self, ind: FinancialIndicator) -> dict[str, Any]:
        return {
            'source_text': ind.source_text,
            'amount': float(ind.amount),
            'currency': ind.currency,
            'fiscal_year': ind.fiscal_year,
```

```
            'funding_source': ind.funding_source,
            'risk_level': ind.risk_level
        }

    # ============================================================================
    # IDENTIFICACIÃ\223N DE RESPONSABLES
    # ============================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.identify_responsible_entities")
    def identify_responsible_entities(self, text: str, tables: list[ExtractedTable]) -> list[ResponsibleEntity]:
        print("ð\237\221¥ Identificando entidades responsables...")

        entities_ner = self._extract_entities_ner(text)
        entities_syntax = self._extract_entities_syntax(text)
        entities_tables = self._extract_from_responsibility_tables(tables)

        all_entities = entities_ner + entities_syntax + entities_tables
        unique_entities = self._consolidate_entities(all_entities)
        scored_entities = self._score_entity_specificity(unique_entities, text)

        print(f" â\234\223 {len(scored_entities)} entidades responsables identificadas")
        return sorted(scored_entities, key=lambda x: x.specificity_score, reverse=True)

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_entities_ner")
    def _extract_entities_ner(self, text: str) -> list[ResponsibleEntity]:
        entities = []
        max_length = 512
        words = text.split()
        chunks = [' '.join(words[i:i + max_length]) for i in range(0, len(words), max_length)]

        for chunk in chunks[:10]:
            try:
                ner_results = self.entity_classifier(chunk)
                for entity in ner_results:
                    if entity['entity_group'] in ['ORG', 'PER'] and entity['score'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_entities_ner", "auto_param_L804_86", 0.7):
                        entities.append(ResponsibleEntity(
                            name=entity['word'],
                            entity_type='secretarÃ-a',
                            specificity_score=entity['score'],
                            mentioned_count=1,
                            associated_programs=[],
                            associated_indicators=[],
                            budget_allocated=None
                        ))
            except Exception:
                continue

        return entities

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_entities_syntax")
    def _extract_entities_syntax(self, text: str) -> list[ResponsibleEntity]:
        entities = []
        responsibility_patterns = [
            r'(?:responsable|ejecutor|encargado|a\s+cargo)[:\s]+([A-ZÃ\201-Ã\232][^\.\n]{10,100})',
            r'(?:secretar[Ã-i]a|direcci[Ã³o]n|oficina)\s+(?:de\s+)?([A-ZÃ\201-Ã\232][^\.\n]{5,80})',
```

```
            r'([A-ZÃ\201-Ã\232][^\.\n]{10,100})\s+(?:ser[Ã¡a]|estar[Ã¡a]|tendr[Ã¡a])\s+(?:responsable|a cargo)'
        ]

        for pattern in responsibility_patterns:
            for match in re.finditer(pattern, text, re.MULTILINE):
                name = match.group(1).strip()
                if len(name) < 10 or len(name) > 150:
                    continue

                entity_type = self._classify_entity_type(name)
                entities.append(ResponsibleEntity(
                    name=name,
                    entity_type=entity_type,
                    specificity_score=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_entities_s
yntax", "auto_param_L838_38", 0.6),
                    mentioned_count=1,
                    associated_programs=[],
                    associated_indicators=[],
                    budget_allocated=None
                ))

        return entities

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._classify_entity_type")
    def _classify_entity_type(self, name: str) -> str:
        name_lower = name.lower()
        if 'secretarÃ-a' in name_lower or 'secretaria' in name_lower:
            return 'secretarÃ-a'
        elif 'direcciÃ³n' in name_lower:
            return 'direcciÃ³n'
        elif 'oficina' in name_lower:
            return 'oficina'
        elif 'alcaldÃ-a' in name_lower or 'alcalde' in name_lower:
            return 'alcaldÃ-a'
        else:
            return 'externo'

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables")
    def _extract_from_responsibility_tables(self, tables: list[ExtractedTable]) -> list[ResponsibleEntity]:
        entities = []
        resp_tables = [t for t in tables if t.table_type == 'responsables']

        for table in resp_tables:
            df = table.df
            resp_cols = [col for col in df.columns if any(
                kw in str(col).lower() for kw in ['responsable', 'ejecutor', 'encargado']
            )]

            if not resp_cols:
                continue

            resp_col = resp_cols[0]
            for value in df[resp_col].dropna().unique():
                name = str(value).strip()
                if len(name) < 5:
                    continue
```

```

                entities.append(ResponsibleEntity(
                    name=name,
                    entity_type=self._classify_entity_type(name),
                    specificity_score=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables", "auto_param_L884_38", 0.8),
                    mentioned_count=1,
                    associated_programs=[],
                    associated_indicators=[],
                    budget_allocated=None
                ))

        return entities

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._consolidate_entities")
    def _consolidate_entities(self, entities: list[ResponsibleEntity]) -> list[ResponsibleEntity]:
        if not entities:
            return []

        names = [e.name for e in entities]
        embeddings = self.semantic_model.encode(names, convert_to_tensor=True)

        similarity_threshold = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._consolidate_entities", "similarity_threshold", 0.85) # Refactored
        clustering = AgglomerativeClustering(
            n_clusters=None,
            distance_threshold=1 - similarity_threshold,
            metric='cosine',
            linkage='average'
        )
        labels = clustering.fit_predict(embeddings.cpu().numpy())

        consolidated = []
        for cluster_id in set(labels):
            cluster_entities = [e for i, e in enumerate(entities) if labels[i] == cluster_id]
            best_entity = max(cluster_entities, key=lambda e: (len(e.name), e.specificity_score, e.mentioned_count))
            total_mentions = sum(e.mentioned_count for e in cluster_entities)

            consolidated.append(ResponsibleEntity(
                name=best_entity.name,
                entity_type=best_entity.entity_type,
                specificity_score=best_entity.specificity_score,
                mentioned_count=total_mentions,
                associated_programs=best_entity.associated_programs,
                associated_indicators=best_entity.associated_indicators,
                budget_allocated=best_entity.budget_allocated
            ))

        return consolidated

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity")
    def _score_entity_specificity(self, entities: list[ResponsibleEntity], full_text: str) -> list[ResponsibleEntity]:
        scored = []
        for entity in entities:
            doc = self.nlp(entity.name)

```

```
            length_score = min(len(entity.name.split()) / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L934_62", 1.0))
            propn_count = sum(1 for token in doc if token.pos_ == ’PROPN’)
            propn_score = min(propn_count / 3, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L936_47", 1.0))

            institutional_words = [’secretarÃ-a’, ’direcciÃ³n’, ’oficina’, ’departamento’, ’coordinaciÃ³n’, ’gerencia’,
                                   ’subdirecciÃ³n’]
            inst_score = float(any(word in entity.name.lower() for word in institutional_words))
            mention_score = min(entity.mentioned_count / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L941_61", 1.0))

            final_score = (length_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L943_42", 0.2) + propn_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L943_62", 0.3) + inst_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L943_81", 0.3) + mention_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity", "auto_param_L943_103", 0.2))

            entity.specificity_score = final_score
            scored.append(entity)

        return scored

    # ========================================================================
    # INFERENCIA CAUSAL - DAG CONSTRUCTION
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.construct_causal_dag")
    def construct_causal_dag(self, text: str, tables: list[ExtractedTable],
                             financial_analysis: dict[str, Any]) -> CausalDAG:
        print("ð\237\224\227 Construyendo grafo causal (DAG)...")

        nodes = self._identify_causal_nodes(text, tables, financial_analysis)
        print(f" â\234\223 {len(nodes)} nodos causales identificados")

        edges = self._identify_causal_edges(text, nodes)
        print(f" â\234\223 {len(edges)} relaciones causales detectadas")

        G = nx.DiGraph()
        for node_name, node in nodes.items():
            G.add_node(node_name, **{
                ’type’: node.node_type,
                ’budget’: float(node.associated_budget) if node.associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.construct_causal_dag", "auto_param_L969_87", 0.0),
                ’evidence’: node.evidence_strength
            })

        for edge in edges:
            if edge.probability > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.construct_causal_dag", "auto_param_L974_34", 0.3):
                G.add_edge(edge.source, edge.target, **{
                    ’type’: edge.edge_type,
                    ’mechanism’: edge.mechanism,
                    ’probability’: edge.probability
                })

```

```
        if not nx.is_directed_acyclic_graph(G):
            print(" â\232 ï¸\217 Detectados ciclos - aplicando topological sorting...")
            G = self._break_cycles(G)

        node_list = list(nodes.keys())
        n = len(node_list)
        adj_matrix = np.zeros((n, n))
        for i, source in enumerate(node_list):
            for j, target in enumerate(node_list):
                if G.has_edge(source, target):
                    adj_matrix[i, j] = G[source][target]['probability']

        print(f" â\234\223 DAG construido: {G.number_of_nodes()} nodos, {G.number_of_edges()} aristas")

        return CausalDAG(nodes=nodes, edges=edges, adjacency_matrix=adj_matrix, graph=G)

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_nodes")
    def _identify_causal_nodes(self, text: str, tables: list[ExtractedTable], financial_analysis: dict[str, Any]) -> \
            dict[str, CausalNode]:
        nodes = {}

        for pillar in self.context.PDET_PILLARS:
            pillar_embedding = self.pdet_embeddings[pillar]
            mentions = self._find_semantic_mentions(text, pillar, pillar_embedding)

            if len(mentions) > 0:
                budget = self._extract_budget_for_pillar(pillar, text, financial_analysis)

                nodes[pillar] = CausalNode(
                    name=pillar,
                    node_type='pilar',
                    embedding=pillar_embedding,
                    associated_budget=budget,
                    temporal_lag=self.context.PDET_THEORY_OF_CHANGE[pillar]['lag_years'],
                    evidence_strength=min(len(mentions) / 5, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_nodes", "auto_param_L1015_61", 1.0))
                )

        for pillar, theory in self.context.PDET_THEORY_OF_CHANGE.items():
            if pillar not in nodes:
                continue

            for outcome in theory['outcomes']:
                outcome_mentions = self._find_outcome_mentions(text, outcome)
                if len(outcome_mentions) > 0:
                    nodes[outcome] = CausalNode(
                        name=outcome,
                        node_type='outcome',
                        embedding=self.semantic_model.encode(outcome, convert_to_tensor=False),
                        associated_budget=None,
                        temporal_lag=0,
                        evidence_strength=min(len(outcome_mentions) / 3, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_nodes", "auto_param_L1031_73", 1.0))
                    )

            for mediator in theory['mediators']:
```

```
                mediator_mentions = self._find_mediator_mentions(text, mediator)
                if len(mediator_mentions) > 0:
                    nodes[mediator] = CausalNode(
                        name=mediator,
                        node_type=’mediator’,
                        embedding=self.semantic_model.encode(mediator, convert_to_tensor=False),
                        associated_budget=None,
                        temporal_lag=0,
                        evidence_strength=min(len(mediator_mentions) / 2, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_nodes", "auto_param_L1043_74", 1.0))
                    )

        return nodes

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_semantic_mentions")
    def _find_semantic_mentions(self, text: str, concept: str, concept_embedding: np.ndarray) -> list[str]:
        sentences = [s.text for s in self.nlp(text[:50000]).sents]

        mentions = []
        for sentence in sentences:
            if len(sentence.split()) < 5:
                continue

            sent_embedding = self.semantic_model.encode(sentence, convert_to_tensor=False)
            similarity = np.dot(concept_embedding, sent_embedding) / (
                    np.linalg.norm(concept_embedding) * np.linalg.norm(sent_embedding)
            )

            if similarity > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_semantic_mentions", "auto_param_L1062_28", 0.5):
                mentions.append(sentence)

        return mentions

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_outcome_mentions")
    def _find_outcome_mentions(self, text: str, outcome: str) -> list[str]:
        outcome_keywords = {
            ’seguridad_juridica’: [’seguridad jurÃ-dica’, ’formalizaciÃ³n’, ’tÃ-tulos’, ’propiedad’],
            ’reduccion_conflictos_tierra’: [’conflicto’, ’tierra’, ’disputa’, ’territorial’],
            ’conectividad’: [’conectividad’, ’vÃ-as’, ’acceso’, ’transporte’],
            ’productividad_agricola’: [’productividad’, ’agrÃ-cola’, ’producciÃ³n’, ’rendimiento’],
            ’mortalidad_infantil’: [’mortalidad infantil’, ’niÃ±os’, ’salud infantil’],
            ’esperanza_vida’: [’esperanza de vida’, ’longevidad’, ’salud’],
            ’cobertura_educativa’: [’cobertura educativa’, ’acceso educaciÃ³n’, ’matrÃ-cula’],
            ’calidad_educativo’: [’calidad educativa’, ’aprendizaje’, ’pruebas saber’],
            ’deficit_habitacional’: [’dÃ©ficit habitacional’, ’vivienda’, ’hogares’],
            ’enfermedades_hidricas’: [’enfermedades hÃ-dricas’, ’agua potable’, ’saneamiento’],
            ’ingreso_rural’: [’ingreso rural’, ’pobreza rural’, ’economÃ-a campesina’],
            ’empleo_rural’: [’empleo rural’, ’trabajo campo’, ’ocupaciÃ³n’],
            ’seguridad_alimentaria’: [’seguridad alimentaria’, ’hambre’, ’nutriciÃ³n’],
            ’nutricion_infantil’: [’nutriciÃ³n infantil’, ’desnutriciÃ³n’, ’alimentaciÃ³n niÃ±os’],
            ’cohesion_social’: [’cohesiÃ³n social’, ’tejido social’, ’comunidad’],
            ’confianza_institucional’: [’confianza’, ’instituciones’, ’legitimidad’]
        }

        keywords = outcome_keywords.get(outcome, [outcome])
```

```
        text_lower = text.lower()

        mentions = []
        for keyword in keywords:
            if keyword in text_lower:
                pattern = f'.{{0,100}}{re.escape(keyword)}.{{0,100}}'
                matches = re.finditer(pattern, text_lower, re.IGNORECASE)
                mentions.extend([m.group() for m in matches])

        return mentions[:10]

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_mediator_mentions")
    def _find_mediator_mentions(self, text: str, mediator: str) -> list[str]:
        mediator_patterns = {
            'formalizacion': ['formalizaciÃ³n', 'titulaciÃ³n', 'escrituras'],
            'acceso_justicia': ['acceso justicia', 'juzgados', 'defensorÃa'],
            'vias_terciarias': ['vÃas terciarias', 'caminos', 'carreteras'],
            'distritos_riego': ['distritos riego', 'irrigaciÃ³n', 'agua agrÃcola'],
            'cobertura_salud': ['cobertura salud', 'eps', 'atenciÃ³n mÃ©dica'],
            'infraestructura_salud': ['hospital', 'centro salud', 'puesto salud'],
            'infraestructura_escolar': ['escuela', 'colegio', 'infraestructura educativa'],
            'docentes_calificados': ['docentes', 'maestros', 'profesores'],
            'cobertura_acueducto': ['acueducto', 'agua potable', 'tuberÃa'],
            'viviendas_dignas': ['vivienda digna', 'casa', 'hogar'],
            'credito_rural': ['crÃ©dito rural', 'financiamiento', 'banco agrario'],
            'asistencia_tecnica': ['asistencia tÃ©cnica', 'extensiÃ³n rural', 'asesorÃa'],
            'produccion_local': ['producciÃ³n local', 'cultivos', 'agricultura'],
            'acceso_alimentos': ['acceso alimentos', 'mercado', 'distribuciÃ³n'],
            'espacios_participacion': ['participaciÃ³n', 'comitÃ©s', 'juntas'],
            'justicia_transicional': ['justicia transicional', 'vÃctimas', 'reparaciÃ³n']
        }

        patterns = mediator_patterns.get(mediator, [mediator])
        text_lower = text.lower()

        mentions = []
        for pattern in patterns:
            if pattern in text_lower:
                matches = re.finditer(f'.{{0,80}}{re.escape(pattern)}.{{0,80}}', text_lower)
                mentions.extend([m.group() for m in matches])

        return mentions[:8]

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_budget_for_pillar")
    def _extract_budget_for_pillar(self, pillar: str, text: str, financial_analysis: dict[str, Any]) -> Decimal | None:
        pillar_lower = pillar.lower()

        for indicator in financial_analysis.get('financial_indicators', []):
            try:
                source_start = text.lower().find(indicator['source_text'].lower())
                if source_start == -1:
                    continue

                context_start = max(0, source_start - 500)
                context_end = min(len(text), source_start + 500)
                context = text[context_start:context_end].lower()
```

```
                if pillar_lower in context:
                    return Decimal(str(indicator[’amount’]))
            except Exception:
                continue

        return None

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges")
    def _identify_causal_edges(self, text: str, nodes: dict[str, CausalNode]) -> list[CausalEdge]:
        edges = []

        for pillar, theory in self.context.PDET_THEORY_OF_CHANGE.items():
            if pillar not in nodes:
                continue

            for mediator in theory[’mediators’]:
                if mediator in nodes:
                    edges.append(CausalEdge(
                        source=pillar,
                        target=mediator,
                        edge_type=’direct’,
                        mechanism="Mecanismo segÃ°n teorÃ-a PDET",
                        probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.8) # Refactored
                    ))

            for outcome in theory[’outcomes’]:
                if outcome in nodes:
                    for mediator in theory[’mediators’]:
                        if mediator in nodes:
                            edges.append(CausalEdge(
                                source=mediator,
                                target=outcome,
                                edge_type=’mediated’,
                                mechanism=f"Mediado por {mediator}",
                                probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.7) # Refactored
                            ))

        causal_patterns = [
            (r’(.+?)\s+(?:genera|produce|causa|lleva a|resulta en|permite)\s+(.+?)[\.\,]’, ’direct’),
            (r’(.+?)\s+mediante\s+(.+?)\s+(?:se logra|alcanza|obtiene)\s+’, ’mediated’),
            (r’para\s+(?:lograr|alcanzar)\s+(.+?)\s+se requiere\s+(.+?)[\.\,]’, ’direct’)
        ]

        for pattern, edge_type in causal_patterns:
            for match in re.finditer(pattern, text[:30000], re.IGNORECASE):
                source_text = match.group(1).strip()
                target_text = match.group(2).strip() if match.lastindex >= 2 else ""

                source_node = self._match_text_to_node(source_text, nodes)
                target_node = self._match_text_to_node(target_text, nodes)

                if source_node and target_node and source_node != target_node:
                    existing = next((e for e in edges if e.source == source_node and e.target == target_node), None)
```

```python

                    if existing:
                        existing.probability = min(existing.probability + ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges", "auto_param_L1201_74", 0.2), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges", "auto_param_L1201_79", 1.0))
                        existing.evidence_quotes.append(match.group(0)[:200])
                    else:
                        edges.append(CausalEdge(
                            source=source_node,
                            target=target_node,
                            edge_type=edge_type,
                            mechanism=match.group(0)[:200],
                            evidence_quotes=[match.group(0)[:200]],
                            probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.6) # Refactored
                        ))

        edges = self._refine_edge_probabilities(edges, text, nodes)

        return edges

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._match_text_to_node")
    def _match_text_to_node(self, text: str, nodes: dict[str, CausalNode]) -> str | None:
        if len(text) < 5:
            return None

        text_embedding = self.semantic_model.encode(text, convert_to_tensor=False)

        best_match = None
        best_similarity = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._match_text_to_node", "best_similarity", 0.0) # Refactored

        for node_name, node in nodes.items():
            if node.embedding is None:
                continue

            similarity = np.dot(text_embedding, node.embedding) / (
                    np.linalg.norm(text_embedding) * np.linalg.norm(node.embedding) + 1e-10
            )

            if similarity > best_similarity and similarity > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._match_text_to_node", "auto_param_L1235_61", 0.4):
                best_similarity = similarity
                best_match = node_name

        return best_match

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities")
    def _refine_edge_probabilities(self, edges: list[CausalEdge], text: str, nodes: dict[str, CausalNode]) -> list[
        CausalEdge]:
        text_lower = text.lower()

        for edge in edges:
            text_lower.count(edge.source[:30].lower())
            text_lower.count(edge.target[:30].lower())

```

```
            cooccurrence_count = 0
            positions_source = [m.start() for m in re.finditer(re.escape(edge.source[:30].lower()), text_lower)]
            positions_target = [m.start() for m in re.finditer(re.escape(edge.target[:30].lower()), text_lower)]

            for pos_s in positions_source:
                for pos_t in positions_target:
                    if abs(pos_s - pos_t) < 500:
                        cooccurrence_count += 1

            if cooccurrence_count > 0:
                boost = min(cooccurrence_count * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1260_49", 0.1), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1260_54", 0.3))
                edge.probability = min(edge.probability + boost, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1261_65", 1.0))

        return edges

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._break_cycles")
    def _break_cycles(self, G: nx.DiGraph) -> nx.DiGraph:
        while not nx.is_directed_acyclic_graph(G):
            try:
                cycle = nx.find_cycle(G)
                weakest_edge = min(cycle, key=lambda e: G[e[0]][e[1]].get('probability', ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._break_cycles", "auto_param_L1270_89", 0.5)))
                G.remove_edge(weakest_edge[0], weakest_edge[1])
            except nx.NetworkXNoCycle:
                break

        return G

    # ========================================================================
    # ESTIMACIÃ\223N BAYESIANA DE EFECTOS CAUSALES
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.estimate_causal_effects")
    def estimate_causal_effects(self, dag: CausalDAG, text: str, financial_analysis: dict[str, Any]) -> list[
        CausalEffect]:
        print("ð\237\223\210 Estimando efectos causales bayesianos...")

        effects = []
        G = dag.graph

        for source in dag.nodes:
            if dag.nodes[source].node_type != 'pilar':
                continue

            reachable_outcomes = [
                node for node, data in G.nodes(data=True)
                if data.get('type') == 'outcome' and nx.has_path(G, source, node)
            ]

            for outcome in reachable_outcomes:
                effect = self._estimate_effect_bayesian(source, outcome, dag, financial_analysis)

                if effect:
```

```
                    effects.append(effect)

        print(f" â\234\223 {len(effects)} efectos causales estimados")
        return effects

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian")
    def _estimate_effect_bayesian(self, treatment: str, outcome: str, dag: CausalDAG,
                                  financial_analysis: dict[str, Any]) -> CausalEffect | None:
        G = dag.graph
        try:
            all_paths = list(nx.all_simple_paths(G, treatment, outcome, cutoff=4))
        except (nx.NetworkXNoPath, nx.NodeNotFound):
            return None

        if not all_paths:
            return None

        [p for p in all_paths if len(p) == 2]
        indirect_paths = [p for p in all_paths if len(p) > 2]

        confounders = self._identify_confounders(treatment, outcome, dag)

        treatment_node = dag.nodes[treatment]
        budget_value = float(treatment_node.associated_budget) if treatment_node.associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1325_104", 0.0)

        with pm.Model():
            prior_mean, prior_sd = self._get_prior_effect(treatment, outcome)

            direct_effect = pm.StudentT('direct_effect', nu=3, mu=prior_mean, sigma=prior_sd)

            indirect_effects = []
            for path in indirect_paths[:3]:
                path_name = '->'.join([p[:15] for p in path])
                indirect_eff = pm.Normal(f'indirect_{path_name}', mu=prior_mean * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1335_82", 0.5), sigma=prior_sd * 1.5)
                indirect_effects.append(indirect_eff)

            if budget_value > 0:
                budget_adjustment = pm.Deterministic('budget_adjustment', pm.math.log1p(budget_value / 1e9))
                adjusted_direct = direct_effect * (1 + budget_adjustment * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1340_75", 0.1))
            else:
                adjusted_direct = direct_effect

            if indirect_effects:
                total_effect = pm.Deterministic('total_effect', adjusted_direct + pm.math.sum(indirect_effects))
            else:
                total_effect = pm.Deterministic('total_effect', adjusted_direct)

            evidence_strength = treatment_node.evidence_strength * dag.nodes[outcome].evidence_strength
            obs_noise = pm.HalfNormal('obs_noise', sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1350_57", 0.5))

            pm.Normal('pseudo_obs', mu=total_effect, sigma=obs_noise,
                                   observed=np.array([evidence_strength * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunic
```

```
ipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1353_74", 0.5)]))

            trace = pm.sample(1500, tune=800, cores=1, return_inferencedata=True, progressbar=False, target_accept=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1355_115", 0.9))

        total_samples = trace.posterior[’total_effect’].values.flatten()
        trace.posterior[’direct_effect’].values.flatten()

        total_mean = float(np.mean(total_samples))
        total_ci = tuple(float(x) for x in np.percentile(total_samples, [2.5, 97.5]))
        prob_positive = float(np.mean(total_samples > 0))
        prob_significant = float(np.mean(np.abs(total_samples) > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1363_65", 0.1)))

        return CausalEffect(
            treatment=treatment,
            outcome=outcome,
            effect_type=’total’,
            point_estimate=float(np.median(total_samples)),
            posterior_mean=total_mean,
            credible_interval_95=total_ci,
            probability_positive=prob_positive,
            probability_significant=prob_significant,
            mediating_paths=indirect_paths,
            confounders_adjusted=confounders
        )

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect")
    def _get_prior_effect(self, treatment: str, outcome: str) -> tuple[float, float]:
        """
        Priors informados basados en meta-anÃ¡lisis de programas PDET
        Referencia: Cinelli et al. (2022) - Sensitivity Analysis for Causal Inference
        """
        effect_priors = {
            (’Infraestructura y adecuaciÃ³n de tierras’, ’productividad_agricola’): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1385_84", 0.35), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1385_90", 0.15)),
            (’Salud rural’, ’mortalidad_infantil’): (-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1386_54", 0.28), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1386_60", 0.12)),
            (’EducaciÃ³n rural y primera infancia’, ’cobertura_educativa’): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1387_76", 0.42), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1387_82", 0.18)),
            (’Vivienda, agua potable y saneamiento bÃ¡sico’, ’enfermedades_hidricas’): (-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1388_88", 0.33), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1388_94", 0.14)),
            (’ReactivaciÃ³n econÃ³mica y producciÃ³n agropecuaria’, ’ingreso_rural’): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1389_84", 0.29), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1389_90", 0.16)),
            (’Sistema para la garantÃa progresiva del derecho a la alimentaciÃ³n’, ’seguridad_alimentaria’): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1390_109", 0.38),
                                                                                                                                                ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1391_108", 0.17)),
        }

        if (treatment, outcome) in effect_priors:
```

```
            return effect_priors[(treatment, outcome)]

        return (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1397_16", 0.2), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1397_21", 0.25))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_confounders")
    def _identify_confounders(self, treatment: str, outcome: str, dag: CausalDAG) -> list[str]:
        """
        Identifica confounders usando d-separation (Pearl, 2009)
        """
        G = dag.graph
        confounders = []

        for node in G.nodes():
            if node in (treatment, outcome):
                continue

            if G.has_edge(node, treatment) and G.has_edge(node, outcome):
                confounders.append(node)

        return confounders

    # ========================================================================
    # ANÃ\201LISIS CONTRAFACTUAL (Pearl's Three-Layer Causal Hierarchy)
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_counterfactuals")
    def generate_counterfactuals(self, dag: CausalDAG, causal_effects: list[CausalEffect],
                                 financial_analysis: dict[str, Any]) -> list[CounterfactualScenario]:
        """
        Genera escenarios contrafactuales usando el framework de Pearl (2009)
        Level 3 - Counterfactual: "What if we had done X instead of Y?"

        ImplementaciÃ³n basada en:
        - Pearl & Mackenzie (2018) - The Book of Why
        - Sharma & Kiciman (2020) - DoWhy: An End-to-End Library for Causal Inference
        """
        print("ð\237\224® Generando escenarios contrafactuales...")

        scenarios = []
        G = dag.graph
        pillar_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'pilar']

        current_budgets = {
            node: float(dag.nodes[node].associated_budget) if dag.nodes[node].associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_counterfactuals", "auto_param_L1438_101", 0.0)
            for node in pillar_nodes
        }
        total_budget = sum(current_budgets.values())

        if total_budget == 0:
            print(" â\232 ï¸\217 No hay informaciÃ³n presupuestal para contrafactuales")
            return scenarios

        # Escenario 1: Incremento proporcional del 20%
        intervention_1 = {node: budget * 1.2 for node, budget in current_budgets.items()}
```

```
        scenario_1 = self._simulate_intervention(intervention_1, dag, causal_effects, "Incremento 20% presupuesto")
        scenarios.append(scenario_1)

        # Escenario 2: Rebalanceo hacia educaciÃ³n y salud
        priority_pillars = ['EducaciÃ³n rural y primera infancia', 'Salud rural']
        intervention_2 = current_budgets.copy()
        for pillar in priority_pillars:
            if pillar in intervention_2:
                intervention_2[pillar] *= 1.5

        other_reduction = (sum(intervention_2.values()) - total_budget) / max(
            len(intervention_2) - len(priority_pillars), 1)
        for pillar in intervention_2:
            if pillar not in priority_pillars:
                intervention_2[pillar] = max(intervention_2[pillar] - other_reduction, 0)

        scenario_2 = self._simulate_intervention(intervention_2, dag, causal_effects,
                                                  "PriorizaciÃ³n educaciÃ³n y salud")
        scenarios.append(scenario_2)

        # Escenario 3: FocalizaciÃ³n en pilar de mayor impacto
        if causal_effects:
            best_effect = max(causal_effects, key=lambda e: e.probability_positive * abs(e.posterior_mean))
            best_pillar = best_effect.treatment

            intervention_3 = {node: budget * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_counterfactuals", "auto_param_L1474_45", 0.7) for node, budget in current_budgets.items()}
            if best_pillar in intervention_3:
                intervention_3[best_pillar] = current_budgets[best_pillar] * 1.8

            scenario_3 = self._simulate_intervention(intervention_3, dag, causal_effects,
                                                      f"FocalizaciÃ³n en {best_pillar[:40]}")
            scenarios.append(scenario_3)

        print(f" â\234\223 {len(scenarios)} escenarios contrafactuales generados")
        return scenarios

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention")
    def _simulate_intervention(self, intervention: dict[str, float], dag: CausalDAG,
                               causal_effects: list[CausalEffect], description: str) -> CounterfactualScenario:
        """
        Simula intervenciÃ³n usando do-calculus (Pearl, 2009)
        Implementa: P(Y | do(X=x)) mediante propagaciÃ³n por el DAG
        """
        G = dag.graph
        predicted_outcomes = {}

        outcome_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'outcome']

        for outcome in outcome_nodes:
            relevant_effects = [e for e in causal_effects if e.outcome == outcome]

            if not relevant_effects:
                continue

            expected_change = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention", "e
```

```
xpected_change", 0.0) # Refactored
            variance_sum = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention", "variance_sum", 0.0) # Refactored

            for effect in relevant_effects:
                treatment = effect.treatment
                if treatment not in intervention:
                    continue

                current_budget = float(dag.nodes[treatment].associated_budget) if dag.nodes[
                    treatment].associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention", "auto_param_L1512_54", 0.0)
                new_budget = intervention[treatment]

                budget_multiplier = new_budget / current_budget if current_budget > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention", "auto_param_L1515_91", 1.0)

                # Rendimientos decrecientes: log transform
                effect_multiplier = np.log1p(budget_multiplier) / np.log1p(ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention", "auto_param_L1518_75", 1.0))

                expected_change += effect.posterior_mean * effect_multiplier

                ci_width = effect.credible_interval_95[1] - effect.credible_interval_95[0]
                variance_sum += (ci_width / 3.92) ** 2  # 95% CI â\211\210 3.92 std

            predicted_std = np.sqrt(variance_sum)
            predicted_outcomes[outcome] = (
                expected_change,
                expected_change - 1.96 * predicted_std,
                expected_change + 1.96 * predicted_std
            )

        probability_improvement = {}
        for outcome, (mean, lower, upper) in predicted_outcomes.items():
            scale = (upper - lower) / 3.92
            if scale <= 0: scale = 1e-9
            prob_positive = stats.norm.sf(0, loc=mean, scale=scale)
            probability_improvement[outcome] = float(prob_positive)

        narrative = self._generate_scenario_narrative(description, intervention, predicted_outcomes,
                                                      probability_improvement)

        return CounterfactualScenario(
            intervention=intervention,
            predicted_outcomes=predicted_outcomes,
            probability_improvement=probability_improvement,
            narrative=narrative
        )

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_scenario_narrative")
    def _generate_scenario_narrative(self, description: str, intervention: dict[str, float],
                                     predicted_outcomes: dict[str, tuple[float, float, float]],
                                     probabilities: dict[str, float]) -> str:
        """Genera narrativa interpretable del escenario contrafactual"""

```

```
        narrative = f"**{description}**\n\n"
        narrative += "**IntervenciÃ³n propuesta:**\n"

        total_intervention = sum(intervention.values())
        for pillar, budget in sorted(intervention.items(), key=lambda x: -x[1])[:5]:
            percentage = (budget / total_intervention * 100) if total_intervention > 0 else 0
            narrative += f"- {pillar[:50]}: ${budget:,.0f} COP ({percentage:.1f}%)\n"

        narrative += "\n**Efectos esperados:**\n"

        significant_outcomes = [(o, p) for o, p in probabilities.items() if p > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_scenario_narrative", "auto_param_L1565_80", 0.6)]
        significant_outcomes.sort(key=lambda x: -x[1])

        for outcome, prob in significant_outcomes[:5]:
            mean, lower, upper = predicted_outcomes[outcome]
            narrative += f"- {outcome}: {mean:+.2f} (IC95%: [{lower:.2f}, {upper:.2f}]) - "
            narrative += f"Probabilidad de mejora: {prob * 100:.0f}%\n"

        return narrative

    # ========================================================================
    # ANÃ\201LISIS DE SENSIBILIDAD (Cinelli et al., 2022)
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.sensitivity_analysis")
    def sensitivity_analysis(self, causal_effects: list[CausalEffect], dag: CausalDAG) -> dict[str, Any]:
        """
        AnÃ¡lisis de sensibilidad para supuestos de identificaciÃ³n causal
        Basado en: Cinelli, Forney & Pearl (2022) - "A Crash Course in Good and Bad Controls"
        """
        print("ð\237\224¬ Ejecutando anÃ¡lisis de sensibilidad...")

        sensitivity_results = {}

        for effect in causal_effects[:10]:  # Top 10 effects
            unobserved_confounding = self._compute_e_value(effect)

            robustness_value = self._compute_robustness_value(effect, dag)

            sensitivity_results[f"{effect.treatment[:30]}â\206\222{effect.outcome[:30]}"] = {
                'e_value': unobserved_confounding,
                'robustness_value': robustness_value,
                'interpretation': self._interpret_sensitivity(unobserved_confounding, robustness_value)
            }

        print(f" â\234\223 Sensibilidad analizada para {len(sensitivity_results)} efectos")
        return sensitivity_results

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_e_value")
    def _compute_e_value(self, effect: CausalEffect) -> float:
        """
        E-value: mÃ-nima fuerza de confounding no observado para anular el efecto
        FÃ³rmula: E = effect_estimate + sqrt(effect_estimate * (effect_estimate - 1))

        Referencia: VanderWeele & Ding (2017) - Ann Intern Med
```

```
        """
        if effect.posterior_mean <= 0:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_e_value", "auto_param_L1612_19", 1.0)

        rr = np.exp(effect.posterior_mean)  # Convert log-scale to risk ratio
        if rr <= 1:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_e_value", "auto_param_L1616_19", 1.0)
        e_value = rr + np.sqrt(rr * (rr - 1))

        return float(e_value)

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value")
    def _compute_robustness_value(self, effect: CausalEffect, dag: CausalDAG) -> float:
        """
        Robustness Value: percentil de la distribuciÃ³n posterior que cruza cero
        Valores altos (>ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value", "auto_param_L1625_24", 0.95)) indican alta robustez
        """
        ci_lower, ci_upper = effect.credible_interval_95

        if ci_lower > 0 or ci_upper < 0:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value", "auto_param_L1630_19", 1.0)

        width = ci_upper - ci_lower
        if width == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value", "auto_param_L1634_19", 0.5)

        robustness = abs(effect.posterior_mean) / (width / 2)
        return float(min(robustness, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value", "auto_param_L1637_37", 1.0)))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity")
    def _interpret_sensitivity(self, e_value: float, robustness: float) -> str:
        """InterpretaciÃ³n de resultados de sensibilidad"""

        if e_value > 2.0 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity", "auto_param_L1643_42", 0.8):
            return "Efecto robusto - Resistente a confounding no observado"
        elif e_value > 1.5 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity", "auto_param_L1645_44", 0.6):
            return "Efecto moderadamente robusto - PrecauciÃ³n con confounders"
        elif e_value > 1.2 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity", "auto_param_L1647_44", 0.4):
            return "Efecto sensible - Alta vulnerabilidad a confounding"
        else:
            return "Efecto frÃ¡gil - Resultados no confiables sin ajustes adicionales"

    # ========================================================================
    # SCORING INTEGRAL DE CALIDAD
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score")
```

```
    def calculate_quality_score(self, text: str, tables: list[ExtractedTable],
                                financial_analysis: dict[str, Any],
                                responsible_entities: list[ResponsibleEntity],
                                causal_dag: CausalDAG,
                                causal_effects: list[CausalEffect]) -> QualityScore:
        """
        Puntaje bayesiano integral de calidad del PDM
        Integra todas las dimensiones de anÃ¡lisis con pesos calibrados
        """
        print("â-\220 Calculando score integral de calidad...")

        financial_score = self._score_financial_component(financial_analysis)
        indicator_score = self._score_indicators(tables, text)
        responsibility_score = self._score_responsibility_clarity(responsible_entities)
        temporal_score = self._score_temporal_consistency(text, tables)
        pdet_score = self._score_pdet_alignment(text, tables, causal_dag)
        causal_score = self._score_causal_coherence(causal_dag, causal_effects)

        weights = np.array([ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_28", 0.20), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_34", 0.15), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_40", 0.15), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_46", 0.10), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_52", 0.20), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_58", 0.20)])
        scores = np.array([
            financial_score, indicator_score, responsibility_score,
            temporal_score, pdet_score, causal_score
        ])

        overall_score = float(np.dot(weights, scores))

        confidence = self._estimate_score_confidence(scores, weights)

        evidence = {
            ’financial’: financial_score,
            ’indicators’: indicator_score,
            ’responsibility’: responsibility_score,
            ’temporal’: temporal_score,
            ’pdet_alignment’: pdet_score,
            ’causal_coherence’: causal_score
        }

        print(f" â\234\223 Score final: {overall_score:.2f}/10.0")

        return QualityScore(
            overall_score=overall_score,
            financial_feasibility=financial_score,
            indicator_quality=indicator_score,
            responsibility_clarity=responsibility_score,
            temporal_consistency=temporal_score,
            pdet_alignment=pdet_score,
            causal_coherence=causal_score,
            confidence_interval=confidence,
            evidence=evidence
        )
```

```python
    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_component")
    def _score_financial_component(self, financial_analysis: dict[str, Any]) -> float:
        """Score componente financiero (0-10)"""

        budget = financial_analysis.get('total_budget', 0)
        if budget == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_component", "auto_param_L1714_19", 0.0)

        budget_score = min(np.log10(float(budget)) / 12, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_component", "auto_param_L1716_57", 1.0)) * 3.0

        diversity = financial_analysis['funding_sources'].get('diversity_index', 0)
        max_diversity = financial_analysis['funding_sources'].get('max_diversity', 1)
        diversity_score = (diversity / max(max_diversity, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_component", "auto_param_L1720_58", 0.1))) * 3.0

        sustainability = financial_analysis.get('sustainability_score', 0)
        sustainability_score = sustainability * 2.5

        risk = financial_analysis['risk_assessment'].get('risk_score', ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_component", "auto_param_L1725_71", 0.5))
        risk_score = (1 - risk) * 1.5

        return float(min(budget_score + diversity_score + sustainability_score + risk_score, 1.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators")
    def _score_indicators(self, tables: list[ExtractedTable], text: str) -> float:
        """Score calidad de indicadores (0-10)"""

        indicator_tables = [t for t in tables if t.table_type == 'indicadores']

        if not indicator_tables:
            baseline_mentions = len(re.findall(r'l[Ã-i]nea\s+base', text, re.IGNORECASE))
            meta_mentions = len(re.findall(r'meta', text, re.IGNORECASE))

            if baseline_mentions > 5 and meta_mentions > 5:
                return 4.0
            return 2.0

        completeness_score = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators", "completeness_score", 0.0) # Refactored
        for table in indicator_tables:
            df = table.df
            required_cols = ['indicador', 'lÃ-nea base', 'meta', 'fuente']
            present_cols = sum(1 for col in required_cols if any(col in str(c).lower() for c in df.columns))
            completeness_score += (present_cols / len(required_cols)) * 3.0

        completeness_score = min(completeness_score, 4.0)

        smart_patterns = [
            r'\d+%',  # Percentages
            r'\d+\s+(?:personas|hogares|familias|hectÃ¡reas)',  # Quantities
            r'reducir|aumentar|mejorar|incrementar',  # Action verbs
        ]

```

```
        smart_count = sum(len(re.findall(pattern, text, re.IGNORECASE)) for pattern in smart_patterns)
        smart_score = min(smart_count / 50, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators", "auto_param_L1760_44", 1.0)) * 3.0

        formula_mentions = len(re.findall(r'f[Ã³o]rmula', text, re.IGNORECASE))
        periodicity_mentions = len(re.findall(r'periodicidad|trimestral|anual|mensual', text, re.IGNORECASE))

        technical_score = min((formula_mentions + periodicity_mentions) / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators", "auto_param_L1765_78", 1.0)) * 3.0

        return float(min(completeness_score + smart_score + technical_score, 10.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_responsibility_clarity")
    def _score_responsibility_clarity(self, entities: list[ResponsibleEntity]) -> float:
        """Score claridad de responsables (0-10)"""

        if not entities:
            return 2.0

        count_score = min(len(entities) / 15, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_responsibility_clarity", "auto_param_L1776_46", 1.0)) * 3.0

        avg_specificity = np.mean([e.specificity_score for e in entities])
        specificity_score = avg_specificity * 4.0

        institutional_entities = [e for e in entities if e.entity_type in ['secretarÃa', 'direcciÃ³n', 'oficina']]
        institutional_ratio = len(institutional_entities) / max(len(entities), 1)
        institutional_score = institutional_ratio * 3.0

        return float(min(count_score + specificity_score + institutional_score, 10.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_temporal_consistency")
    def _score_temporal_consistency(self, text: str, tables: list[ExtractedTable]) -> float:
        """Score consistencia temporal (0-10)"""

        years_mentioned = set(re.findall(r'20[2-3]\d', text))

        if len(years_mentioned) < 2:
            return 3.0

        years = [int(y) for y in years_mentioned]
        year_range = max(years) - min(years) if years else 0
        range_score = min(year_range / 4, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_temporal_consistency", "auto_param_L1798_42", 1.0)) * 3.0

        cronograma_tables = [t for t in tables if t.table_type == 'cronograma']
        cronograma_score = min(len(cronograma_tables) * 2, 4.0)

        temporal_terms = ['cronograma', 'aÃ±o', 'trimestre', 'mes', 'periodo', 'etapa', 'fase']
        term_count = sum(len(re.findall(rf'\b{term}\b', text, re.IGNORECASE)) for term in temporal_terms)
        term_score = min(term_count / 30, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_temporal_consistency", "auto_param_L1805_42", 1.0)) * 3.0

        return float(min(range_score + cronograma_score + term_score, 10.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_pdet_alignment")
```

```
    def _score_pdet_alignment(self, text: str, tables: list[ExtractedTable], dag: CausalDAG) -> float:
        """Score alineaciÃ³n con pilares PDET (0-10)"""

        text_lower = text.lower()

        pillar_mentions = {}
        for pillar in self.context.PDET_PILLARS:
            pillar_lower = pillar.lower()
            keywords = pillar_lower.split()[:3]

            count = sum(text_lower.count(kw) for kw in keywords)
            pillar_mentions[pillar] = count

        coverage = sum(1 for count in pillar_mentions.values() if count > 0)
        coverage_score = (coverage / len(self.context.PDET_PILLARS)) * 4.0

        pdet_explicit = len(re.findall(r’\bPDET\b’, text, re.IGNORECASE))
        patr_mentions = len(re.findall(r’\bPATR\b’, text, re.IGNORECASE))
        explicit_score = min((pdet_explicit + patr_mentions) / 15, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_pdet_alignment", "auto_param_L1828_67", 1.0)) * 3.0

        pdet_tables = [t for t in tables if t.table_type == ’pdet’]
        table_score = min(len(pdet_tables) * 1.5, 3.0)

        return float(min(coverage_score + explicit_score + table_score, 10.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence")
    def _score_causal_coherence(self, dag: CausalDAG, effects: list[CausalEffect]) -> float:
        """Score coherencia causal del plan (0-10)"""

        G = dag.graph

        if G.number_of_nodes() == 0:
            return 2.0

        structure_score = min(G.number_of_edges() / (G.number_of_nodes() * 1.5), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence", "auto_param_L1844_81", 1.0)) * 3.0

        if not effects:
            effect_quality = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence", "effect_quality", 0.0) # Refactored
        else:
            avg_probability = np.mean([e.probability_significant for e in effects])
            effect_quality = avg_probability * 4.0

        pillar_nodes = [n for n, data in G.nodes(data=True) if data.get(’type’) == ’pilar’]
        outcome_nodes = [n for n, data in G.nodes(data=True) if data.get(’type’) == ’outcome’]

        connected_pillars = sum(1 for p in pillar_nodes if any(nx.has_path(G, p, o) for o in outcome_nodes))
        connectivity = (connected_pillars / max(len(pillar_nodes), 1)) * 3.0

        return float(min(structure_score + effect_quality + connectivity, 10.0))

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_score_confidence")
    def _estimate_score_confidence(self, scores: np.ndarray, weights: np.ndarray) -> tuple[float, float]:
        """Estima intervalo de confianza para el score usando bootstrap"""
```

```

        n_bootstrap = 1000
        bootstrap_scores = []

        for _ in range(n_bootstrap):
            noise = np.random.normal(0, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_score_confidence", "auto_param_L1868_40", 0.5), size=len(scores))
            noisy_scores = np.clip(scores + noise, 0, 10)

            bootstrap_score = np.dot(weights, noisy_scores)
            bootstrap_scores.append(bootstrap_score)

        ci_lower, ci_upper = np.percentile(bootstrap_scores, [2.5, 97.5])

        return (float(ci_lower), float(ci_upper))

    # ========================================================================
    # EXPORTACIÃ\223N Y VISUALIZACIÃ\223N
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.export_causal_network")
    def export_causal_network(self, dag: CausalDAG, output_path: str) -> None:
        """Exporta el DAG causal en formato GraphML para Gephi/Cytoscape"""

        G = dag.graph.copy()

        for node, data in G.nodes(data=True):
            data['label'] = node[:50]
            data['node_type'] = data.get('type', 'unknown')
            data['budget'] = data.get('budget', ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.export_causal_network", "auto_param_L1891_48", 0.0))

        for _u, _v, data in G.edges(data=True):
            data['weight'] = data.get('probability', ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.export_causal_network", "auto_param_L1894_53", 0.5))
            data['edge_type'] = data.get('type', 'unknown')

        nx.write_graphml(G, output_path)
        print(f"â\234\205 Red causal exportada a: {output_path}")

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_executive_report")
    def generate_executive_report(self, analysis_results: dict[str, Any]) -> str:
        """Genera reporte ejecutivo en Markdown"""

        report = "# ANÃ\201LISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET\n\n"
        report += f"**Fecha de anÃ¡lisis:** {datetime.now().strftime('%Y-%m-%d %H:%M')}\n\n"

        report += "## 1. RESUMEN EJECUTIVO\n\n"

        quality = analysis_results['quality_score']
        report += f"**Score Global de Calidad:** {quality['overall_score']:.2f}/10.0 "
        report += f"(IC95%: [{quality['confidence_interval'][0]:.2f}, {quality['confidence_interval'][1]:.2f}])\n\n"

        report += self._interpret_overall_quality(quality['overall_score'])
        report += "\n\n"

```

```
        report += "### Dimensiones Evaluadas\n\n"
        report += f"- **Viabilidad Financiera:** {quality['financial_feasibility']:.1f}/10\n"
        report += f"- **Calidad de Indicadores:** {quality['indicator_quality']:.1f}/10\n"
        report += f"- **Claridad de Responsables:** {quality['responsibility_clarity']:.1f}/10\n"
        report += f"- **Consistencia Temporal:** {quality['temporal_consistency']:.1f}/10\n"
        report += f"- **AlineaciÃ³n PDET:** {quality['pdet_alignment']:.1f}/10\n"
        report += f"- **Coherencia Causal:** {quality['causal_coherence']:.1f}/10\n\n"

        report += "## 2. ANÃ\201LISIS FINANCIERO\n\n"
        fin = analysis_results['financial_analysis']
        report += f"**Presupuesto Total:** ${fin['total_budget']:,.0f} COP\n\n"

        report += "### DistribuciÃ³n por Fuente\n\n"
        if fin['funding_sources'] and fin['funding_sources']['distribution']:
            for source, amount in sorted(fin['funding_sources']['distribution'].items(), key=lambda x: -x[1])[:5]:
                pct = (amount / fin['total_budget'] * 100) if fin['total_budget'] > 0 else 0
                report += f"- {source}: ${amount:,.0f} ({pct:.1f}%)\n"

        report += f"\n**Ã\215ndice de DiversificaciÃ³n:** {fin['funding_sources'].get('diversity_index', 0):.2f}\n"
        report += f"**Score de Sostenibilidad:** {fin['sustainability_score']:.2f}\n"
        report += f"**EvaluaciÃ³n de Riesgo:** {fin['risk_assessment']['interpretation']}\n\n"

        report += "## 3. INFERENCIA CAUSAL\n\n"

        effects = analysis_results.get('causal_effects', [])
        if effects:
            report += "### Efectos Causales Principales\n\n"

            significant_effects = [e for e in effects if e['probability_significant'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_executive_report", "auto_param_L1944_88", 0.7)]
            significant_effects.sort(key=lambda e: abs(e['posterior_mean']), reverse=True)

            for effect in significant_effects[:5]:
                report += f"**{effect['treatment'][:40]} â\206\222 {effect['outcome'][:40]}**\n"
                report += f"- Efecto estimado: {effect['posterior_mean']:+.3f} "
                report += f"(IC95%: [{effect['credible_interval'][0]:.3f}, {effect['credible_interval'][1]:.3f}])\n"
                report += f"- Probabilidad de efecto positivo: {effect['probability_positive'] * 100:.0f}%\n"

                if effect['mediating_paths']:
                    report += f"- VÃ-as de mediaciÃ³n: {len(effect['mediating_paths'])}\n"
                report += "\n"

        report += "## 4. ESCENARIOS CONTRAFACTUALES\n\n"

        scenarios = analysis_results.get('counterfactuals', [])
        for _i, scenario in enumerate(scenarios, 1):
            report += scenario['narrative']
            report += "\n---\n\n"

        report += "## 5. ANÃ\201LISIS DE SENSIBILIDAD\n\n"

        sensitivity = analysis_results.get('sensitivity_analysis', {})
        if sensitivity:
            report += "| RelaciÃ³n Causal | E-Value | Robustez | InterpretaciÃ³n |\n"
            report += "|-----------------|---------|----------|-----------------|\n"

```

```
            for relation, metrics in list(sensitivity.items())[:8]:
                report += f"| {relation} | {metrics[’e_value’]:.2f} | {metrics[’robustness_value’]:.2f} | {metrics[’interpretation’][:50]} |\n"

        report += "\n## 6. RECOMENDACIONES\n\n"
        report += self._generate_recommendations(analysis_results)

        report += "\n---\n\n"
        report += "*AnÃ¡lisis generado por PDETMunicipalPlanAnalyzer v5.0*\n"
        report += "*MetodologÃa: Inferencia Causal Bayesiana + Structural Causal Models*\n"

        return report

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_overall_quality")
    def _interpret_overall_quality(self, score: float) -> str:
        """InterpretaciÃ³n del score global"""

        if score >= 8.0:
            return ("**EvaluaciÃ³n: EXCELENTE** â\234\205\n\n"
                    "El plan cumple con altos estÃ¡ndares de calidad tÃ©cnica. "
                    "Presenta coherencia causal sÃ³lida, viabilidad financiera demostrable, "
                    "y alineaciÃ³n robusta con los pilares PDET.")
        elif score >= 6.5:
            return ("**EvaluaciÃ³n: BUENO** â\234\223\n\n"
                    "El plan presenta bases sÃ³lidas pero con oportunidades de mejora. "
                    "Se recomienda fortalecer algunos componentes especÃficos.")
        elif score >= 5.0:
            return ("**EvaluaciÃ³n: ACEPTABLE** â\232 ï¸\217\n\n"
                    "El plan cumple requisitos mÃnimos pero requiere ajustes sustanciales "
                    "en mÃºltiples dimensiones para asegurar efectividad.")
        else:
            return ("**EvaluaciÃ³n: DEFICIENTE** â\235\214\n\n"
                    "El plan presenta deficiencias crÃticas que comprometen su viabilidad. "
                    "Se requiere reformulaciÃ³n integral.")

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations")
    def _generate_recommendations(self, analysis_results: dict[str, Any]) -> str:
        """Genera recomendaciones especÃficas basadas en el anÃ¡lisis"""

        recommendations = []
        quality = analysis_results[’quality_score’]

        # Recomendaciones financieras
        if quality[’financial_feasibility’] < 6.0:
            fin = analysis_results[’financial_analysis’]
            if fin[’funding_sources’].get(’dependency_risk’, 0) > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations", "auto_param_L2015_66", 0.6):
                recommendations.append(
                    "**DiversificaciÃ³n de fuentes:** Reducir dependencia excesiva de fuentes Ãºnicas. "
                    "Explorar alternativas como cooperaciÃ³n internacional, APP, o gestiÃ³n de recursos propios."
                )

            if fin[’sustainability_score’] < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations", "auto_param_L2021_45", 0.5):
                recommendations.append(
                    "**Sostenibilidad fiscal:** Fortalecer componente de recursos propios. "
                    "Desarrollar estrategias de generaciÃ³n de ingresos municipales."
```

```
            )

        # Recomendaciones de indicadores
        if quality['indicator_quality'] < 6.0:
            recommendations.append(
                "**Fortalecimiento de indicadores:** Definir indicadores SMART completos "
                "(especÃ-ficos, medibles, alcanzables, relevantes, temporales) con lÃ-neas base, "
                "metas cuantificadas, fÃ³rmulas de cÃ¡lculo y fuentes verificables."
            )

        # Recomendaciones causales
        effects = analysis_results.get('causal_effects', [])
        if effects:
            weak_effects = [e for e in effects if e['probability_significant'] < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations", "auto_param_L2038_81", 0.5)]

            if len(weak_effects) > len(effects) * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations", "auto_param_L2040_50", 0.5):
                recommendations.append(
                    "**Robustez causal:** Fortalecer vÃ-nculos entre intervenciones y resultados esperados. "
                    "Explicitar teorÃ-as de cambio y mecanismos causales subyacentes."
                )

        # Recomendaciones PDET
        if quality['pdet_alignment'] < 6.0:
            recommendations.append(
                "**AlineaciÃ³n PDET:** Articular explÃ-citamente con los 8 pilares del Pacto Estructurante. "
                "Referenciar iniciativas PATR y asegurar coherencia con transformaciÃ³n territorial."
            )

        # Recomendaciones de responsabilidad
        if quality['responsibility_clarity'] < 6.0:
            recommendations.append(
                "**Claridad institucional:** Especificar responsables concretos para cada programa. "
                "Evitar asignaciones genÃ©ricas como 'todas las secretarÃ-as' o 'alcaldÃ-a municipal'."
            )

        # Recomendaciones de mejores escenarios
        scenarios = analysis_results.get('counterfactuals', [])
        if scenarios:
            best_scenario = max(scenarios,
                                key=lambda s: sum(s['probability_improvement'].values()))

            recommendations.append(
                f"**OptimizaciÃ³n presupuestal:** Considerar escenario '{best_scenario['narrative'].split('**')[1]}' "
                "que maximiza probabilidad de impacto en outcomes clave."
            )

        if not recommendations:
            return "El plan presenta solidez en todas las dimensiones evaluadas. Continuar con implementaciÃ³n segÃºn lo planificado.\n"

        result = ""
        for i, rec in enumerate(recommendations, 1):
            result += f"{i}. {rec}\n\n"

        return result
```

```

    # ========================================================================
    # PIPELINE PRINCIPAL
    # ========================================================================

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.analyze_municipal_plan_sync")
    def analyze_municipal_plan_sync(self, pdf_path: str, output_dir: str | None = None) -> dict[str, Any]:
        """Synchronous wrapper for analyze_municipal_plan."""

        loop = asyncio.new_event_loop()
        try:
            return loop.run_until_complete(self.analyze_municipal_plan(pdf_path, output_dir))
        finally:
            loop.close()

    async def analyze_municipal_plan(self, pdf_path: str, output_dir: str | None = None) -> dict[str, Any]:
        """
        Pipeline completo de anÃ¡lisis

        Args:
            pdf_path: Ruta al PDF del Plan de Desarrollo Municipal
            output_dir: Directorio para guardar outputs (opcional)

        Returns:
            Diccionario con todos los resultados del anÃ¡lisis
        """

        print("\n" + "=" * 70)
        print("ANÃ\201LISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET")
        print("=" * 70 + "\n")

        start_time = datetime.now()

        # 1. ExtracciÃ³n de texto
        print("ð\237\223\204 Extrayendo texto del PDF...")
        full_text = self._extract_full_text(pdf_path)
        print(f" â\234\223 {len(full_text)} caracteres extraÃ-dos\n")

        # 2. ExtracciÃ³n de tablas
        tables = await self.extract_tables(pdf_path)

        # 3. AnÃ¡lisis financiero
        financial_analysis = self.analyze_financial_feasibility(tables, full_text)

        # 4. IdentificaciÃ³n de responsables
        responsible_entities = self.identify_responsible_entities(full_text, tables)

        # 5. ConstrucciÃ³n de DAG causal
        causal_dag = self.construct_causal_dag(full_text, tables, financial_analysis)

        # 6. EstimaciÃ³n de efectos causales
        causal_effects = self.estimate_causal_effects(causal_dag, full_text, financial_analysis)

        # 7. GeneraciÃ³n de contrafactuales
        counterfactuals = self.generate_counterfactuals(causal_dag, causal_effects, financial_analysis)

```

```
        # 8. AnÃ¡lisis de sensibilidad
        sensitivity_analysis = self.sensitivity_analysis(causal_effects, causal_dag)

        # 9. Score integral de calidad
        quality_score = self.calculate_quality_score(
            full_text, tables, financial_analysis, responsible_entities,
            causal_dag, causal_effects
        )

        # 10. CompilaciÃ³n de resultados
        results = {
            ’metadata’: {
                ’pdf_path’: pdf_path,
                ’analysis_date’: datetime.now().isoformat(),
                ’processing_time_seconds’: (datetime.now() - start_time).total_seconds(),
                ’analyzer_version’: ’5.0’
            },
            ’extraction’: {
                ’text_length’: len(full_text),
                ’tables_extracted’: len(tables),
                ’table_types’: {t.table_type: sum(1 for x in tables if x.table_type == t.table_type)
                                for t in tables if t.table_type}
            },
            ’financial_analysis’: financial_analysis,
            ’responsible_entities’: [self._entity_to_dict(e) for e in responsible_entities[:20]],
            ’causal_dag’: {
                ’nodes’: len(causal_dag.nodes),
                ’edges’: len(causal_dag.edges),
                ’pillar_nodes’: [n for n, node in causal_dag.nodes.items() if node.node_type == ’pilar’],
                ’outcome_nodes’: [n for n, node in causal_dag.nodes.items() if node.node_type == ’outcome’]
            },
            ’causal_effects’: [self._effect_to_dict(e) for e in causal_effects[:15]],
            ’counterfactuals’: [self._scenario_to_dict(s) for s in counterfactuals],
            ’sensitivity_analysis’: sensitivity_analysis,
            ’quality_score’: self._quality_to_dict(quality_score)
        }

        # 11. ExportaciÃ³n de resultados
        if output_dir:
            output_path = Path(output_dir)
            output_path.mkdir(parents=True, exist_ok=True)

            # Exportar DAG
            dag_path = output_path / "causal_network.graphml"
            self.export_causal_network(causal_dag, str(dag_path))

            # Exportar reporte
            # Delegate to factory for I/O operation
            from farfan_pipeline.analysis.factory import save_json, write_text_file

            report = self.generate_executive_report(results)
            report_path = output_path / "executive_report.md"
            write_text_file(report, report_path)
            print(f"â\234\205 Reporte ejecutivo guardado en: {report_path}")

            # Exportar JSON
```

```
            json_path = output_path / "analysis_results.json"
            save_json(results, json_path)
            print(f"â\234\205 Resultados JSON guardados en: {json_path}")

        elapsed = (datetime.now() - start_time).total_seconds()
        print(f"\nâ\217±ï¸\217 AnÃ¡lisis completado en {elapsed:.1f} segundos")
        print("=" * 70 + "\n")

        return results

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_full_text")
    def _extract_full_text(self, pdf_path: str) -> str:
        """Extrae texto completo del PDF usando mÃºltiples mÃ©todos"""

        text_parts = []

        # MÃ©todo 1: PyMuPDF (rÃ¡pido y eficiente)
        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import open_pdf_with_fitz, open_pdf_with_pdfplumber

        try:
            doc = open_pdf_with_fitz(pdf_path)
            for page in doc:
                text_parts.append(page.get_text())
            doc.close()
        except Exception as e:
            print(f" â\232 ï¸\217 PyMuPDF fallÃ³: {str(e)[:50]}")

        # MÃ©todo 2: pdfplumber (mejor para tablas complejas)
        try:
            pdf = open_pdf_with_pdfplumber(pdf_path)
            for page in pdf.pages[:100]:  # LÃmite de 100 pÃ¡ginas
                text = page.extract_text()
                if text:
                    text_parts.append(text)
            pdf.close()
        except Exception as e:
            print(f" â\232 ï¸\217 pdfplumber fallÃ³: {str(e)[:50]}")

        full_text = ’\n\n’.join(text_parts)

        # Limpieza bÃ¡sica
        full_text = re.sub(r’\n{3,}’, ’\n\n’, full_text)
        full_text = re.sub(r’ {2,}’, ’ ’, full_text)

        return full_text

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._entity_to_dict")
    def _entity_to_dict(self, entity: ResponsibleEntity) -> dict[str, Any]:
        """Convierte ResponsibleEntity a diccionario"""
        return {
            ’name’: entity.name,
            ’type’: entity.entity_type,
            ’specificity_score’: entity.specificity_score,
            ’mentions’: entity.mentioned_count,
            ’programs’: entity.associated_programs,
```

```
            'budget': float(entity.budget_allocated) if entity.budget_allocated else None
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._effect_to_dict")
    def _effect_to_dict(self, effect: CausalEffect) -> dict[str, Any]:
        """Convierte CausalEffect a diccionario"""
        return {
            'treatment': effect.treatment,
            'outcome': effect.outcome,
            'effect_type': effect.effect_type,
            'point_estimate': effect.point_estimate,
            'posterior_mean': effect.posterior_mean,
            'credible_interval': effect.credible_interval_95,
            'probability_positive': effect.probability_positive,
            'probability_significant': effect.probability_significant,
            'mediating_paths': effect.mediating_paths,
            'confounders_adjusted': effect.confounders_adjusted
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._scenario_to_dict")
    def _scenario_to_dict(self, scenario: CounterfactualScenario) -> dict[str, Any]:
        """Convierte CounterfactualScenario a diccionario"""
        return {
            'intervention': scenario.intervention,
            'predicted_outcomes': scenario.predicted_outcomes,
            'probability_improvement': scenario.probability_improvement,
            'narrative': scenario.narrative
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._quality_to_dict")
    def _quality_to_dict(self, quality: QualityScore) -> dict[str, Any]:
        """Convierte QualityScore a diccionario"""
        return {
            'overall_score': quality.overall_score,
            'financial_feasibility': quality.financial_feasibility,
            'indicator_quality': quality.indicator_quality,
            'responsibility_clarity': quality.responsibility_clarity,
            'temporal_consistency': quality.temporal_consistency,
            'pdet_alignment': quality.pdet_alignment,
            'causal_coherence': quality.causal_coherence,
            'confidence_interval': quality.confidence_interval,
            'evidence': quality.evidence
        }

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_product_mentions")
    def _find_product_mentions(self, text: str) -> list[str]:
        """
        Find mentions of products in text.

        Args:
            text: Text to search

        Returns:
            List of product mentions
        """
        products = []
```

```

        # Common product keywords
        product_patterns = [
            r'producto\s+(\d+)',
            r'servicio\s+(\d+)',
            r'bien\s+(\d+)',
            r'actividad\s+(\d+)',
        ]

        for pattern in product_patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                products.append(match.group(0))

        # Also look for numbered lists that might be products
        list_pattern = r'^\s*\d+\.\s+([^\n]+)'
        for match in re.finditer(list_pattern, text, re.MULTILINE):
            item_text = match.group(1).lower()
            if any(word in item_text for word in ['producto', 'servicio', 'actividad', 'bien']):
                products.append(match.group(1))

        return products

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_optimal_remediations")
    def _generate_optimal_remediations(self, gaps: list[dict[str, Any]]) -> list[dict[str, str]]:
        """
        Generate optimal remediations for identified gaps.

        Args:
            gaps: List of identified gaps

        Returns:
            List of remediation recommendations
        """
        remediations = []

        for gap in gaps:
            remediation = {
                'gap_type': gap.get('type', 'unknown'),
                'priority': 'high' if gap.get('severity') == 'high' else 'medium',
                'recommendation': ''
            }

            gap_type = gap.get('type', '')

            if gap_type == 'missing_baseline':
                remediation['recommendation'] = "Establecer lÃnea base cuantitativa basada en diagnÃ³stico actual"
            elif gap_type == 'missing_target':
                remediation['recommendation'] = "Definir meta cuantitativa con horizonte temporal claro"
            elif gap_type == 'missing_entity':
                remediation['recommendation'] = "Asignar entidad responsable especÃfica"
            elif gap_type == 'missing_budget':
                remediation['recommendation'] = "Asignar presupuesto especÃfico con fuente de financiaciÃ³n"
            elif gap_type == 'missing_indicator':
                remediation['recommendation'] = "Definir indicador medible con fÃ³rmula de cÃ¡lculo"
            else:
```

```
                remediation['recommendation'] = f"Completar {gap_type} segÃºn estÃ¡ndares DNP"

            remediations.append(remediation)

        return remediations

    @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations")
    def generate_recommendations(self, analysis_results: dict[str, Any]) -> list[str]:
        """
        Generate recommendations based on analysis results.

        Args:
            analysis_results: Results from municipal plan analysis

        Returns:
            List of actionable recommendations
        """
        recommendations = []

        # Check financial feasibility
        if analysis_results.get('financial_feasibility', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2379_62", 0.7):
            recommendations.append(
                "Revisar sostenibilidad financiera y diversificar fuentes de financiaciÃ³n"
            )

        # Check indicator quality
        if analysis_results.get('indicator_quality', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2385_58", 0.7):
            recommendations.append(
                "Mejorar calidad de indicadores: asegurar lÃnea base, meta y fuente de informaciÃ³n"
            )

        # Check responsibility clarity
        if analysis_results.get('responsibility_clarity', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2391_63", 0.7):
            recommendations.append(
                "Clarificar entidades responsables para cada producto y resultado"
            )

        # Check temporal consistency
        if analysis_results.get('temporal_consistency', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2397_61", 0.7):
            recommendations.append(
                "Establecer cronograma claro con hitos y plazos definidos"
            )

        # Check causal coherence
        if analysis_results.get('causal_coherence', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2403_57", 0.7):
            recommendations.append(
                "Fortalecer coherencia causal: vincular productos con resultados e impactos"
            )

        # PDET-specific recommendations
        if analysis_results.get('is_pdet_municipality', False):
```

```
            if analysis_results.get('pdet_alignment', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2410_59", 0.7):
                recommendations.append(
                    "Alinear intervenciones con lineamientos PDET y enfoque territorial"
                )

        # Generic recommendation if no specific issues
        if not recommendations:
            recommendations.append(
                "El plan cumple con estÃ¡ndares mÃnimos. Considerar monitoreo continuo."
            )

        return recommendations

# ============================================================================
# UTILIDADES Y HELPERS
# ============================================================================

class PDETAnalysisException(Exception):
    """ExcepciÃ³n personalizada para errores de anÃ¡lisis"""
    pass

def validate_pdf_path(pdf_path: str) -> Path:
    """Valida que el path del PDF exista y sea vÃ¡lido"""

    path = Path(pdf_path)

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

    if not path.is_file():
        raise PDETAnalysisException(f"La ruta no es un archivo: {pdf_path}")

    if path.suffix.lower() != '.pdf':
        raise PDETAnalysisException(f"El archivo debe ser PDF, encontrado: {path.suffix}")

    return path

def setup_logging(log_level: str = 'INFO') -> None:
    """Configura logging para el anÃ¡lisis"""

    import logging

    logging.basicConfig(
        level=getattr(logging, log_level.upper()),
        format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
        handlers=[
            logging.StreamHandler(),
            logging.FileHandler('pdet_analysis.log', encoding='utf-8')
        ]
    )

# ============================================================================
# EJEMPLO DE USO
# ============================================================================

```

```
async def main_example() -> None:
    """
    Ejemplo de uso del analizador

    REQUISITOS PREVIOS:
    1. Instalar dependencias: pip install -r requirements.txt
    2. Descargar modelo SpaCy: python -m spacy download es_dep_news_trf
    3. Tener GPU disponible (opcional pero recomendado)
    """

    # Configurar logging
    setup_logging(’INFO’)

    # Inicializar analizador
    analyzer = PDETMunicipalPlanAnalyzer(
        use_gpu=True,
        language=’es’,
        confidence_threshold = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations",
"confidence_threshold", 0.7) # Refactored
    )

    # Ruta al PDF del Plan de Desarrollo Municipal
    pdf_path = "path/to/plan_desarrollo_municipal.pdf"

    try:
        # Validar archivo
        validate_pdf_path(pdf_path)

        # Ejecutar anÃ¡lisis completo
        results = await analyzer.analyze_municipal_plan(
            pdf_path=pdf_path,
            output_dir="outputs/analisis_pdm"
        )

        # Acceder a resultados especÃficos
        print("\nð\237\223\212 RESULTADOS PRINCIPALES:")
        print(f" Score de Calidad: {results[’quality_score’][’overall_score’]:.2f}/10")
        print(f" Presupuesto Total: ${results[’financial_analysis’][’total_budget’]:,.0f}")
        print(f" Efectos Causales Identificados: {len(results[’causal_effects’])}")
        print(f" Escenarios Contrafactuales: {len(results[’counterfactuals’])}")

    except PDETAnalysisException as e:
        print(f"â\235\214 Error de anÃ¡lisis: {e}")
    except Exception as e:
        print(f"â\235\214 Error inesperado: {e}")
        raise



================================================================================
FILE: src/farfan_pipeline/analysis/graph_metrics_fallback.py
================================================================================

"""
Graph metrics computation with NetworkX fallback handling.

```

```
This module provides graph metrics computation with graceful degradation
when NetworkX is unavailable. It integrates with the runtime configuration
system to emit proper observability signals.
"""

import logging
from typing import Any, Optional

from farfan_pipeline.core.runtime_config import RuntimeConfig, get_runtime_config
from farfan_pipeline.core.contracts.runtime_contracts import (
    GraphMetricsInfo,
    FallbackCategory,
)
from farfan_pipeline.core.observability.structured_logging import log_fallback
from farfan_pipeline.core.observability.metrics import increment_graph_metrics_skipped

logger = logging.getLogger(__name__)


def check_networkx_available() -> bool:
    """
    Check if NetworkX is available for graph metrics computation.

    Returns:
        True if NetworkX is available, False otherwise
    """
    try:
        import networkx
        return True
    except ImportError:
        return False


def compute_graph_metrics_with_fallback(
    graph_data: Any,
    runtime_config: Optional[RuntimeConfig] = None,
    document_id: Optional[str] = None,
) -> tuple[dict[str, Any], GraphMetricsInfo]:
    """
    Compute graph metrics with NetworkX fallback handling.

    Args:
        graph_data: Graph data structure (e.g., edge list, adjacency matrix)
        runtime_config: Optional runtime configuration (uses global if None)
        document_id: Optional document identifier for logging

    Returns:
        Tuple of (metrics_dict, GraphMetricsInfo manifest)

    Example:
        >>> metrics, info = compute_graph_metrics_with_fallback(edge_list)
        >>> if info.computed:
        ...     print(f"Centrality: {metrics['centrality']}")
        ... else:
        ...     print(f"Skipped: {info.reason}")
    """
```

```
    if runtime_config is None:
        runtime_config = get_runtime_config()

    networkx_available = check_networkx_available()

    if networkx_available:
        try:
            import networkx as nx

            # Convert graph_data to NetworkX graph
            # This is a placeholder - actual implementation depends on graph_data format
            if isinstance(graph_data, list):
                # Assume edge list format: [(source, target), ...]
                G = nx.Graph()
                G.add_edges_from(graph_data)
            elif isinstance(graph_data, dict):
                # Assume adjacency dict format
                G = nx.from_dict_of_lists(graph_data)
            else:
                raise ValueError(f"Unsupported graph_data type: {type(graph_data)}")

            # Compute graph metrics
            metrics = {
                'num_nodes': G.number_of_nodes(),
                'num_edges': G.number_of_edges(),
                'density': nx.density(G),
                'avg_clustering': nx.average_clustering(G) if G.number_of_nodes() > 0 else 0.0,
                'num_components': nx.number_connected_components(G),
            }

            # Compute centrality if graph is not too large
            if G.number_of_nodes() < 1000:
                metrics['degree_centrality'] = nx.degree_centrality(G)
                metrics['betweenness_centrality'] = nx.betweenness_centrality(G)

            logger.info(f"Graph metrics computed: {metrics['num_nodes']} nodes, {metrics['num_edges']} edges")

            graph_info = GraphMetricsInfo(
                computed=True,
                networkx_available=True,
                reason=None
            )

            return metrics, graph_info

        except Exception as e:
            # NetworkX available but computation failed
            logger.error(f"Graph metrics computation failed: {e}")

            reason = f"NetworkX computation error: {str(e)}"
            graph_info = GraphMetricsInfo(
                computed=False,
                networkx_available=True,
                reason=reason
            )

```

```
            # Emit structured log and metrics (Category B: Quality degradation)
            log_fallback(
                component='graph_metrics',
                subsystem='analysis',
                fallback_category=FallbackCategory.B,
                fallback_mode='computation_error',
                reason=reason,
                runtime_mode=runtime_config.mode,
                document_id=document_id,
            )

            increment_graph_metrics_skipped(
                reason='computation_error',
                runtime_mode=runtime_config.mode,
            )

            # Return empty metrics
            return {}, graph_info

    else:
        # NetworkX not available - graceful degradation
        reason = "NetworkX not available - graph metrics skipped"
        logger.warning(reason)

        graph_info = GraphMetricsInfo(
            computed=False,
            networkx_available=False,
            reason=reason
        )

        # Emit structured log and metrics (Category B: Quality degradation)
        log_fallback(
            component='graph_metrics',
            subsystem='analysis',
            fallback_category=FallbackCategory.B,
            fallback_mode='networkx_unavailable',
            reason=reason,
            runtime_mode=runtime_config.mode,
            document_id=document_id,
        )

        increment_graph_metrics_skipped(
            reason='networkx_unavailable',
            runtime_mode=runtime_config.mode,
        )

        # Return empty metrics
        return {}, graph_info


def compute_basic_graph_stats(graph_data: Any) -> dict[str, Any]:
    """
    Compute basic graph statistics without NetworkX.

    This is a lightweight fallback that computes basic stats
    without requiring NetworkX.
```

```

    Args:
        graph_data: Graph data (edge list or adjacency dict)

    Returns:
        Dictionary with basic graph statistics
    """
    if isinstance(graph_data, list):
        # Edge list format
        nodes = set()
        for edge in graph_data:
            if len(edge) >= 2:
                nodes.add(edge[0])
                nodes.add(edge[1])

        return {
            ’num_nodes’: len(nodes),
            ’num_edges’: len(graph_data),
            ’method’: ’basic_stats_no_networkx’
        }

    elif isinstance(graph_data, dict):
        # Adjacency dict format
        num_edges = sum(len(neighbors) for neighbors in graph_data.values())

        return {
            ’num_nodes’: len(graph_data),
            ’num_edges’: num_edges // 2,  # Undirected graph
            ’method’: ’basic_stats_no_networkx’
        }

    else:
        return {
            ’num_nodes’: 0,
            ’num_edges’: 0,
            ’method’: ’unknown_format’
        }



================================================================================
FILE: src/farfan_pipeline/analysis/macro_prompts.py
================================================================================

# macro_prompts.py
"""
Macro Prompts for MACRO-Level Analysis
======================================

This module implements 5 strategic macro-level analysis prompts:
1. Coverage & Structural Gap Stressor - Evaluates dimensional/cluster coverage
2. Inter-Level Contradiction Scan - Detects microâ\206\224mesoâ\206\224macro contradictions
3. Bayesian Portfolio Composer - Integrates posteriors into global portfolio
4. Roadmap Optimizer - Generates sequenced 0-3m / 3-6m / 6-12m roadmap
5. Peer Normalization & Confidence Scaling - Adjusts classification vs peers

```

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

import logging
import statistics
import warnings
from dataclasses import asdict, dataclass, field
from typing import Any

from farfan_pipeline.core.parameters import ParameterLoaderV2

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

logger = logging.getLogger(__name__)

# ============================================================================
# DATA STRUCTURES FOR MACRO PROMPTS
# ============================================================================

@dataclass
class CoverageAnalysis:
    """Output from Coverage & Structural Gap Stressor"""
    coverage_index: float  # Weighted average coverage (0.0-1.0)
    degraded_confidence: float | None  # Adjusted confidence if coverage low
    predictive_uplift: dict[str, float]  # Expected improvement if gaps filled
    dimension_coverage: dict[str, float]  # D1-D6 coverage percentages
    policy_area_coverage: dict[str, float]  # P1-P10 coverage percentages
    critical_dimensions_below_threshold: list[str]  # Dimensions needing attention
    metadata: dict[str, Any] = field(default_factory=dict)

@dataclass
class ContradictionReport:
    """Output from Inter-Level Contradiction Scan"""
    contradictions: list[dict[str, Any]]  # List of detected contradictions
    suggested_actions: list[dict[str, str]]  # Actions to resolve contradictions
    consistency_score: float  # 0.0-1.0 overall consistency
    micro_meso_alignment: float  # 0.0-1.0 microâ\206\224meso alignment
    meso_macro_alignment: float  # 0.0-1.0 mesoâ\206\224macro alignment
    metadata: dict[str, Any] = field(default_factory=dict)

@dataclass
class BayesianPortfolio:
    """Output from Bayesian Portfolio Composer"""
    prior_global: float  # Global prior (weighted meso average)
    penalties_applied: dict[str, float]  # Coverage, dispersion, contradiction penalties
    posterior_global: float  # Adjusted global posterior
    var_global: float  # Global variance
    confidence_interval: tuple[float, float]  # 95% CI
    metadata: dict[str, Any] = field(default_factory=dict)

@dataclass
class ImplementationRoadmap:
    """Output from Roadmap Optimizer
```

```

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """
    phases: list[dict[str, Any]]  # 0-3m, 3-6m, 6-12m phases
    total_expected_uplift: float  # Total expected improvement
    critical_path: list[str]  # Critical dependency chain
    resource_requirements: dict[str, Any]  # Estimated resources per phase
    metadata: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self) -> None:
        warnings.warn(
            "ImplementationRoadmap is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )

@dataclass
class PeerNormalization:
    """Output from Peer Normalization & Confidence Scaling

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """
    z_scores: dict[str, float]  # Z-scores by policy area
    adjusted_confidence: float  # Adjusted confidence based on peer comparison
    peer_position: str  # "above_average", "average", "below_average"
    outlier_areas: list[str]  # Policy areas >2 SD from mean
    metadata: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self) -> None:
        warnings.warn(
            "PeerNormalization is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )

# ============================================================================
# MACRO PROMPT 1: COVERAGE & STRUCTURAL GAP STRESSOR
# ============================================================================

class CoverageGapStressor:
```

```
    """
    ROLE: Structural Integrity Auditor [systems design]
    GOAL: Evaluar si la ausencia de clusters o dimensiones erosiona la validez del score macro.

    INPUTS:
    - convergence_by_dimension
    - missing_clusters
    - dimension_coverage: {D1..D6: % preguntas respondidas}
    - policy_area_coverage: {P#: %}

    MANDATES:
    - Calcular coverage_index (media ponderada)
    - Si dimension_coverage < Ï\204 en alguna dimensiÃ³n crÃ-tica (D3, D6) â\206\222 degradar global_confidence
    - Simular impacto si se completara el cluster faltante (predictive uplift)

    OUTPUT:
    JSON {coverage_index, degraded_confidence, predictive_uplift}

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(
        self,
        critical_dimensions: list[str] | None = None,
        dimension_weights: dict[str, float] | None = None,
        coverage_threshold: float = 0.70
    ) -> None:
        """
        Initialize Coverage Gap Stressor

        Args:
            critical_dimensions: List of critical dimensions (default: D3, D6)
            dimension_weights: Weights for each dimension (default: equal)
            coverage_threshold: Minimum acceptable coverage (default: 0.70)
        """
        warnings.warn(
            "CoverageGapStressor is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.critical_dimensions = critical_dimensions or ["D3", "D6"]
        self.dimension_weights = dimension_weights or {
            f"D{i}": 1.0/6.0 for i in range(1, 7)
        }
        self.coverage_threshold = coverage_threshold
        logger.info(f"CoverageGapStressor initialized with threshold={coverage_threshold}")

    def evaluate(
```

```
        self,
        convergence_by_dimension: dict[str, float],
        missing_clusters: list[str],
        dimension_coverage: dict[str, float],
        policy_area_coverage: dict[str, float],
        baseline_confidence: float = 1.0
    ) -> CoverageAnalysis:
        """
        Evaluate coverage and structural gaps

        Args:
            convergence_by_dimension: Convergence scores by dimension
            missing_clusters: List of missing cluster names
            dimension_coverage: Coverage percentage by dimension
            policy_area_coverage: Coverage percentage by policy area
            baseline_confidence: Starting confidence level (0.0-1.0)

        Returns:
            CoverageAnalysis with complete gap assessment
        """
        # Calculate weighted coverage index
        coverage_index = self._calculate_coverage_index(dimension_coverage)

        # Check critical dimensions
        critical_below_threshold = self._identify_critical_gaps(dimension_coverage)

        # Degrade confidence if critical gaps exist
        degraded_confidence = self._degrade_confidence(
            baseline_confidence,
            critical_below_threshold,
            coverage_index
        )

        # Simulate predictive uplift
        predictive_uplift = self._simulate_uplift(
            missing_clusters,
            dimension_coverage,
            convergence_by_dimension
        )

        return CoverageAnalysis(
            coverage_index=coverage_index,
            degraded_confidence=degraded_confidence,
            predictive_uplift=predictive_uplift,
            dimension_coverage=dimension_coverage,
            policy_area_coverage=policy_area_coverage,
            critical_dimensions_below_threshold=critical_below_threshold,
            metadata={
                "missing_clusters": missing_clusters,
                "threshold_used": self.coverage_threshold,
                "critical_dimensions": self.critical_dimensions
            }
        )

    def _calculate_coverage_index(
        self,
```

```
        dimension_coverage: dict[str, float]
    ) -> float:
        """Calculate weighted average coverage index"""
        total_weight = 0.0
        weighted_sum = 0.0

        for dim, coverage in dimension_coverage.items():
            weight = self.dimension_weights.get(dim, 0.0)
            weighted_sum += coverage * weight
            total_weight += weight

        if total_weight == 0:
            return 0.0

        return weighted_sum / total_weight

    def _identify_critical_gaps(
        self,
        dimension_coverage: dict[str, float]
    ) -> list[str]:
        """Identify critical dimensions below threshold"""
        critical_gaps = []

        for dim in self.critical_dimensions:
            if dim in dimension_coverage:
                if dimension_coverage[dim] < self.coverage_threshold:
                    critical_gaps.append(dim)

        return critical_gaps

    def _degrade_confidence(
        self,
        baseline_confidence: float,
        critical_gaps: list[str],
        coverage_index: float
    ) -> float:
        """Degrade confidence based on structural gaps"""
        degraded = baseline_confidence

        # Penalty for each critical gap
        for _ in critical_gaps:
            degraded *= 0.85  # 15% penalty per critical gap

        # Additional penalty if overall coverage is low
        if coverage_index < self.coverage_threshold:
            gap_severity = (self.coverage_threshold - coverage_index) / self.coverage_threshold
            degraded *= (1.0 - gap_severity * 0.3)  # Up to 30% additional penalty

        return max(0.0, min(1.0, degraded))

    def _simulate_uplift(
        self,
        missing_clusters: list[str],
        dimension_coverage: dict[str, float],
        convergence_by_dimension: dict[str, float]
    ) -> dict[str, float]:
```

```
        """Simulate impact if missing clusters were completed"""
        uplift = {}

        # Estimate uplift for each missing cluster
        for cluster in missing_clusters:
            # Assume cluster completion would improve coverage by 10-20%
            estimated_improvement = 0.15
            uplift[cluster] = estimated_improvement

        # Estimate dimension-level uplift
        for dim, coverage in dimension_coverage.items():
            if coverage < 1.0:
                gap = 1.0 - coverage
                convergence = convergence_by_dimension.get(dim, 0.5)
                # Higher convergence suggests more potential uplift
                potential_uplift = gap * convergence * 0.7
                uplift[f"{dim}_completion"] = potential_uplift

        return uplift

# ============================================================================
# MACRO PROMPT 2: INTER-LEVEL CONTRADICTION SCAN
# ============================================================================

class ContradictionScanner:
    """
    ROLE: Consistency Inspector [data governance]
    GOAL: Detectar contradicciones microâ\206\224mesoâ\206\224macro.

    INPUTS:
    - micro_claims (extraÃ-do de MicroLevelAnswer.evidence)
    - meso_summary_signals
    - macro_narratives (borrador)

    MANDATES:
    - Alinear claims por entidad/tema/dimensiÃ³n
    - Marcar contradicciÃ³n si macro afirma X y â\211¥k micro niegan X con posterior â\211¥ Î¸
    - Sugerir correcciÃ³n: "rephrase / downgrade confidence / request re-execution"

    OUTPUT:
    JSON {contradictions[], suggested_actions}

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(
        self,
        contradiction_threshold: int = 3,
        posterior_threshold: float = 0.7
    ) -> None:
        """
```

```
        Initialize Contradiction Scanner

        Args:
            contradiction_threshold: Min number of micro claims to flag contradiction
            posterior_threshold: Min posterior confidence to consider claim valid
        """
        warnings.warn(
            "ContradictionScanner is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.k = contradiction_threshold
        self.theta = posterior_threshold
        logger.info(f"ContradictionScanner initialized (k={self.k}, Î¸={self.theta})")

    def scan(
        self,
        micro_claims: list[dict[str, Any]],
        meso_summary_signals: dict[str, Any],
        macro_narratives: dict[str, Any]
    ) -> ContradictionReport:
        """
        Scan for contradictions across levels

        Args:
            micro_claims: List of micro-level claims with evidence
            meso_summary_signals: Meso-level summary signals
            macro_narratives: Macro-level narrative statements

        Returns:
            ContradictionReport with detected issues and suggested actions
        """
        # Align claims by entity/theme/dimension
        aligned_claims = self._align_claims(micro_claims, meso_summary_signals, macro_narratives)

        # Detect contradictions (defensive: ensure returns list)
        contradictions = ensure_list_return(self._detect_contradictions(aligned_claims))

        # Generate suggested actions
        suggested_actions = self._generate_actions(contradictions)

        # Calculate consistency scores
        consistency_score = self._calculate_consistency(contradictions, len(micro_claims))
        micro_meso_alignment = self._calculate_alignment(micro_claims, meso_summary_signals)
        meso_macro_alignment = self._calculate_alignment(
            [meso_summary_signals],
            macro_narratives
        )

        return ContradictionReport(
            contradictions=contradictions,
            suggested_actions=suggested_actions,
            consistency_score=consistency_score,
```

```
            micro_meso_alignment=micro_meso_alignment,
            meso_macro_alignment=meso_macro_alignment,
            metadata={
                "total_micro_claims": len(micro_claims),
                "contradiction_threshold": self.k,
                "posterior_threshold": self.theta
            }
        )

    def _align_claims(
        self,
        micro_claims: list[dict[str, Any]],
        meso_summary_signals: dict[str, Any],
        macro_narratives: dict[str, Any]
    ) -> dict[str, dict[str, list[Any]]]:
        """Align claims by entity/theme/dimension"""
        aligned = {
            "micro": {},
            "meso": {},
            "macro": {}
        }

        # Group micro claims by dimension
        for claim in micro_claims:
            dimension = claim.get("dimension", "unknown")
            if dimension not in aligned["micro"]:
                aligned["micro"][dimension] = []
            aligned["micro"][dimension].append(claim)

        # Group meso signals by dimension
        for key, value in meso_summary_signals.items():
            if key.startswith("D") and len(key) == 2:
                aligned["meso"][key] = value

        # Group macro narratives
        aligned["macro"] = macro_narratives

        return aligned

    def _detect_contradictions(
        self,
        aligned_claims: dict[str, dict[str, Any]]
    ) -> list[dict[str, Any]]:
        """Detect contradictions across levels"""
        contradictions = []

        # Check each dimension/theme
        for dimension in aligned_claims.get("micro", {}):
            micro_claims = aligned_claims["micro"].get(dimension, [])
            aligned_claims["meso"].get(dimension, {})
            macro_narrative = aligned_claims["macro"].get(dimension, {})

            # Count claims that contradict macro narrative
            contradicting_claims = []
            for claim in micro_claims:
                if self._is_contradictory(claim, macro_narrative):
```

```
                    posterior = claim.get("posterior", 0.0)
                    if posterior >= self.theta:
                        contradicting_claims.append(claim)

            # Flag if threshold exceeded
            if len(contradicting_claims) >= self.k:
                contradictions.append({
                    "dimension": dimension,
                    "type": "micro_macro_contradiction",
                    "contradicting_claims": len(contradicting_claims),
                    "threshold": self.k,
                    "details": contradicting_claims[:5]  # Sample
                })

        return contradictions

    def _is_contradictory(
        self,
        claim: dict[str, Any],
        narrative: dict[str, Any]
    ) -> bool:
        """Check if claim contradicts narrative"""
        # Simple heuristic: if claim score is low but narrative is positive
        claim_score = claim.get("score", 0.0)
        narrative_score = narrative.get("score", 0.5)

        # Contradiction if scores differ significantly
        return abs(claim_score - narrative_score) > 0.4

    def _generate_actions(
        self,
        contradictions: list[dict[str, Any]]
    ) -> list[dict[str, str]]:
        """Generate suggested actions to resolve contradictions"""
        actions = []

        for contradiction in contradictions:
            dimension = contradiction.get("dimension", "unknown")
            count = contradiction.get("contradicting_claims", 0)

            if count >= self.k * 2:
                actions.append({
                    "dimension": dimension,
                    "action": "request_re_execution",
                    "reason": f"{count} micro claims contradict macro narrative"
                })
            elif count >= self.k:
                actions.append({
                    "dimension": dimension,
                    "action": "downgrade_confidence",
                    "reason": f"{count} micro claims suggest lower confidence"
                })
            else:
                actions.append({
                    "dimension": dimension,
                    "action": "rephrase_narrative",
```

```
                    "reason": "Minor inconsistencies detected"
                })

        return actions

    def _calculate_consistency(
        self,
        contradictions: list[dict[str, Any]],
        total_claims: int
    ) -> float:
        """Calculate overall consistency score"""
        if total_claims == 0:
            return 1.0

        total_contradictions = sum(
            c.get("contradicting_claims", 0) for c in contradictions
        )

        consistency = 1.0 - (total_contradictions / max(total_claims, 1))
        return max(0.0, min(1.0, consistency))

    def _calculate_alignment(
        self,
        level1_data: Any,
        level2_data: Any
    ) -> float:
        """Calculate alignment between two levels"""
        # Simplified alignment calculation
        # In production, would use semantic similarity, score correlation, etc.
        return 0.85  # Placeholder

# ============================================================================
# MACRO PROMPT 3: BAYESIAN PORTFOLIO COMPOSER
# ============================================================================

class BayesianPortfolioComposer:
    """
    ROLE: Global Bayesian Integrator [causal inference]
    GOAL: Integrar todas las posteriors (micro y meso) en una cartera causal global.

    INPUTS:
    - meso_posteriors
    - weighting_trace (cluster_weights)
    - macro_reconciliation_penalties

    MANDATES:
    - Calcular prior_global (media ponderada meso)
    - Aplicar penalties jerÃ¡rquicos (coverage, dispersion estructural, contradictions)
    - Recalcular posterior_global y varianza

    OUTPUT:
    JSON {prior_global, penalties_applied, posterior_global, var_global}

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
```

```
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(
        self,
        default_variance: float = 0.05
    ) -> None:
        """
        Initialize Bayesian Portfolio Composer

        Args:
            default_variance: Default variance for uncertain estimates
        """
        warnings.warn(
            "BayesianPortfolioComposer is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.default_variance = default_variance
        logger.info("BayesianPortfolioComposer initialized")

    def compose(
        self,
        meso_posteriors: dict[str, float],
        cluster_weights: dict[str, float],
        reconciliation_penalties: dict[str, float] | None = None
    ) -> BayesianPortfolio:
        """
        Compose global Bayesian portfolio from meso posteriors

        Args:
            meso_posteriors: Posterior probabilities by cluster/dimension
            cluster_weights: Weights for each cluster
            reconciliation_penalties: Optional penalties (coverage, dispersion, contradictions)

        Returns:
            BayesianPortfolio with integrated global estimate
        """
        # Calculate weighted prior
        prior_global = self._calculate_weighted_prior(meso_posteriors, cluster_weights)

        # Apply hierarchical penalties
        penalties = reconciliation_penalties or {}
        penalties_applied = self._apply_penalties(prior_global, penalties)

        # Calculate posterior and variance
        posterior_global = self._calculate_posterior(prior_global, penalties_applied)
        var_global = self._calculate_variance(meso_posteriors, cluster_weights, penalties_applied)

        # Calculate 95% confidence interval
        ci = self._calculate_confidence_interval(posterior_global, var_global)
```

```

        return BayesianPortfolio(
            prior_global=prior_global,
            penalties_applied=penalties_applied,
            posterior_global=posterior_global,
            var_global=var_global,
            confidence_interval=ci,
            metadata={
                "num_clusters": len(meso_posteriors),
                "total_weight": sum(cluster_weights.values())
            }
        )

    def _calculate_weighted_prior(
        self,
        meso_posteriors: dict[str, float],
        cluster_weights: dict[str, float]
    ) -> float:
        """Calculate weighted prior from meso posteriors"""
        total_weight = sum(cluster_weights.values())
        if total_weight == 0:
            return 0.5  # Neutral prior

        weighted_sum = 0.0
        for cluster, posterior in meso_posteriors.items():
            weight = cluster_weights.get(cluster, 0.0)
            weighted_sum += posterior * weight

        return weighted_sum / total_weight

    def _apply_penalties(
        self,
        prior: float,
        penalties: dict[str, float]
    ) -> dict[str, float]:
        """Apply hierarchical penalties"""
        applied = {}

        # Coverage penalty
        coverage_penalty = penalties.get("coverage", 0.0)
        applied["coverage"] = coverage_penalty

        # Structural dispersion penalty
        dispersion_penalty = penalties.get("dispersion", 0.0)
        applied["dispersion"] = dispersion_penalty

        # Contradiction penalty
        contradiction_penalty = penalties.get("contradictions", 0.0)
        applied["contradictions"] = contradiction_penalty

        return applied

    def _calculate_posterior(
        self,
        prior: float,
        penalties: dict[str, float]
```

```
    ) -> float:
        """Calculate posterior after applying penalties"""
        posterior = prior

        # Apply each penalty multiplicatively
        for _penalty_name, penalty_value in penalties.items():
            posterior *= (1.0 - penalty_value)

        return max(0.0, min(1.0, posterior))

    def _calculate_variance(
        self,
        meso_posteriors: dict[str, float],
        cluster_weights: dict[str, float],
        penalties: dict[str, float]
    ) -> float:
        """Calculate global variance"""
        if len(meso_posteriors) < 2:
            return self.default_variance

        # Calculate weighted variance
        mean = self._calculate_weighted_prior(meso_posteriors, cluster_weights)
        total_weight = sum(cluster_weights.values())

        if total_weight == 0:
            return self.default_variance

        weighted_sq_diff = 0.0
        for cluster, posterior in meso_posteriors.items():
            weight = cluster_weights.get(cluster, 0.0)
            sq_diff = (posterior - mean) ** 2
            weighted_sq_diff += weight * sq_diff

        variance = weighted_sq_diff / total_weight

        # Increase variance based on penalties
        penalty_factor = 1.0 + sum(penalties.values())
        adjusted_variance = variance * penalty_factor

        return adjusted_variance

    def _calculate_confidence_interval(
        self,
        posterior: float,
        variance: float,
        confidence: float = 0.95
    ) -> tuple[float, float]:
        """Calculate confidence interval (assumes normal distribution)"""
        # For 95% CI, z-score â\211\210 1.96
        z_score = 1.96
        margin = z_score * (variance ** 0.5)

        lower = max(0.0, posterior - margin)
        upper = min(1.0, posterior + margin)

        return (lower, upper)
```

```

# ============================================================================
# MACRO PROMPT 4: ROADMAP OPTIMIZER
# ============================================================================

class RoadmapOptimizer:
    """
    ROLE: Execution Strategist [operations design]
    GOAL: Generar roadmap secuenciado 0â\200\2233m / 3â\200\2236m / 6â\200\22312m priorizando impacto/costo.

    INPUTS:
    - critical_gaps (list)
    - dependency_graph (gaps con prerequisitos)
    - effort_estimates
    - impact_scores

    MANDATES:
    - Ordenar por ratio impact/effort y dependencias
    - Asignar ventana temporal mÃ-nima sin colisiÃ³n de prerequisitos
    - Estimar uplift esperado por tramo

    OUTPUT:
    JSON roadmap {phase, actions[], expected_uplift}

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(self) -> None:
        """Initialize Roadmap Optimizer"""
        warnings.warn(
            "RoadmapOptimizer is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        logger.info("RoadmapOptimizer initialized")

    def optimize(
        self,
        critical_gaps: list[dict[str, Any]],
        dependency_graph: dict[str, list[str]],
        effort_estimates: dict[str, float],
        impact_scores: dict[str, float]
    ) -> ImplementationRoadmap:
        """
        Generate optimized implementation roadmap

        Args:
            critical_gaps: List of gaps to address
```

```
            dependency_graph: Gap ID -> list of prerequisite gap IDs
            effort_estimates: Gap ID -> effort estimate (person-months)
            impact_scores: Gap ID -> expected impact (ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L702_54", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L702_58", 1.0))

        Returns:
            ImplementationRoadmap with phased action plan
        """
        # Calculate impact/effort ratios
        prioritized_gaps = self._prioritize_gaps(
            critical_gaps,
            effort_estimates,
            impact_scores
        )

        # Assign to time windows respecting dependencies
        phases = self._assign_phases(
            prioritized_gaps,
            dependency_graph,
            effort_estimates
        )

        # Calculate expected uplift per phase
        total_uplift = self._calculate_total_uplift(phases, impact_scores)

        # Identify critical path
        critical_path = self._identify_critical_path(dependency_graph, impact_scores)

        # Estimate resource requirements
        resources = self._estimate_resources(phases, effort_estimates)

        return ImplementationRoadmap(
            phases=phases,
            total_expected_uplift=total_uplift,
            critical_path=critical_path,
            resource_requirements=resources,
            metadata={
                "total_gaps": len(critical_gaps),
                "total_effort": sum(effort_estimates.values())
            }
        )

    def _prioritize_gaps(
        self,
        gaps: list[dict[str, Any]],
        effort_estimates: dict[str, float],
        impact_scores: dict[str, float]
    ) -> list[dict[str, Any]]:
        """Prioritize gaps by impact/effort ratio"""
        prioritized = []

        for gap in gaps:
            gap_id = gap.get("id", "unknown")
            effort = effort_estimates.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L752_50", 1.0))
            impact = impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L753_47", 0
```

```
.5))

            # Calculate ratio (avoid division by zero)
            ratio = impact / max(effort, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L756_41", 0.1))

            prioritized.append({
                **gap,
                "priority_ratio": ratio,
                "effort": effort,
                "impact": impact
            })

        # Sort by priority ratio (descending)
        prioritized.sort(key=lambda x: x["priority_ratio"], reverse=True)

        return prioritized

    def _assign_phases(
        self,
        prioritized_gaps: list[dict[str, Any]],
        dependency_graph: dict[str, list[str]],
        effort_estimates: dict[str, float]
    ) -> list[dict[str, Any]]:
        """Assign gaps to time phases respecting dependencies"""
        phases = [
            {"name": "0-3m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L778_54", 0.0), "max_effort": 9.0},
            {"name": "3-6m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L779_54", 0.0), "max_effort": 9.0},
            {"name": "6-12m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L780_55", 0.0), "max_effort": 18.0}
        ]

        assigned = set()
        gap_dict = {gap.get("id"): gap for gap in prioritized_gaps}

        # Process gaps, but assign dependencies first
        def assign_gap_recursive(gap_id: str, visited: set) -> None:
            """Recursively assign gap and its dependencies"""
            if gap_id in assigned or gap_id in visited:
                return

            visited.add(gap_id)

            # First assign dependencies
            dependencies = dependency_graph.get(gap_id, [])
            for dep_id in dependencies:
                if dep_id in gap_dict:
                    assign_gap_recursive(dep_id, visited)

            # Now assign this gap
            if gap_id not in assigned and gap_id in gap_dict:
                gap = gap_dict[gap_id]
                effort = gap.get("effort", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L803_43", 1.0))

                # Find earliest phase where all dependencies are satisfied
```

```
                earliest_phase = self._get_earliest_phase(dependencies, assigned, phases)

                # Assign to earliest phase with capacity
                for i in range(earliest_phase, len(phases)):
                    if phases[i]["effort"] + effort <= phases[i]["max_effort"]:
                        phases[i]["actions"].append(gap)
                        phases[i]["effort"] += effort
                        assigned.add(gap_id)
                        break

        # Assign gaps in priority order, but respecting dependencies
        for gap in prioritized_gaps:
            gap_id = gap.get("id", "unknown")
            assign_gap_recursive(gap_id, set())

        return phases

    def _get_earliest_phase(
        self,
        dependencies: list[str],
        assigned: set,
        phases: list[dict[str, Any]]
    ) -> int:
        """Get earliest phase where all dependencies are satisfied"""
        if not dependencies:
            return 0

        max_dep_phase = -1
        for dep_id in dependencies:
            # Find which phase the dependency is in
            dep_found = False
            for i, phase in enumerate(phases):
                for action in phase["actions"]:
                    if action.get("id") == dep_id:
                        max_dep_phase = max(max_dep_phase, i)
                        dep_found = True
                        break
                if dep_found:
                    break

        # Return phase after latest dependency (or 0 if no dependencies found yet)
        return min(max_dep_phase + 1, len(phases) - 1) if max_dep_phase >= 0 else 0

    def _calculate_total_uplift(
        self,
        phases: list[dict[str, Any]],
        impact_scores: dict[str, float]
    ) -> float:
        """Calculate total expected uplift across all phases"""
        total = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "total", 0.0) # Refactored

        for phase in phases:
            for action in phase["actions"]:
                gap_id = action.get("id", "unknown")
                impact = impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L860_51", 0.0))
```

```
                total += impact

        return total

    def _identify_critical_path(
        self,
        dependency_graph: dict[str, list[str]],
        impact_scores: dict[str, float]
    ) -> list[str]:
        """Identify critical dependency chain"""
        # Find the path with highest total impact
        # Simple heuristic: find longest chain with high-impact nodes

        # Find nodes with no dependents (endpoints)
        has_dependents = set()
        for deps in dependency_graph.values():
            has_dependents.update(deps)

        endpoints = [
            gap_id for gap_id in dependency_graph
            if gap_id not in has_dependents
        ]

        # For each endpoint, trace back to find highest-impact path
        best_path = []
        best_impact = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "best_impact", 0.0) # Refactored

        for endpoint in endpoints:
            path = self._trace_path(endpoint, dependency_graph)
            path_impact = sum(impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L890_56", 0.0)) for gap_id in path)

            if path_impact > best_impact:
                best_impact = path_impact
                best_path = path

        return best_path

    def _trace_path(
        self,
        gap_id: str,
        dependency_graph: dict[str, list[str]]
    ) -> list[str]:
        """Trace dependency path from gap to root"""
        path = [gap_id]
        dependencies = dependency_graph.get(gap_id, [])

        if dependencies:
            # Follow first dependency (simplified)
            dep_path = self._trace_path(dependencies[0], dependency_graph)
            path = dep_path + path

        return path

    def _estimate_resources(
        self,
```

```
        phases: list[dict[str, Any]],
        effort_estimates: dict[str, float]
    ) -> dict[str, Any]:
        """Estimate resource requirements per phase"""
        resources = {}

        for phase in phases:
            phase_name = phase["name"]
            total_effort = phase["effort"]
            num_actions = len(phase["actions"])

            # Estimate team size (assuming 3 months per person-month per phase)
            phase_months = {"0-3m": 3, "3-6m": 3, "6-12m": 6}
            months = phase_months.get(phase_name, 3)
            team_size = max(1, int(total_effort / months))

            resources[phase_name] = {
                "total_effort_months": total_effort,
                "recommended_team_size": team_size,
                "num_actions": num_actions
            }

        return resources

# ==========================================================================
# MACRO PROMPT 5: PEER NORMALIZATION & CONFIDENCE SCALING
# ==========================================================================

class PeerNormalizer:
    """
    ROLE: Macro Peer Evaluator [evaluation design]
    GOAL: Ajustar clasificaciÃ³n macro considerando comparativos regionales.

    INPUTS:
    - convergence_by_policy_area
    - peer_distributions: {policy_area -> {mean, std}}
    - baseline_confidence

    MANDATES:
    - Calcular z-scores
    - Penalizar si >k Ã¡reas estÃ¡n < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L956_37", 1.0) z
    - Aumentar confianza si todas dentro Â±ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L957_42", 0.5)
z y dispersiÃ³n baja

    OUTPUT:
    JSON {z_scores, adjusted_confidence}

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(
```

```
        self,
        penalty_threshold: int = 3,
        outlier_z_threshold: float = 2.0
    ) -> None:
        """
        Initialize Peer Normalizer

        Args:
            penalty_threshold: Number of low-performing areas to trigger penalty
            outlier_z_threshold: Z-score threshold for outlier identification
        """
        warnings.warn(
            "PeerNormalizer is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.k = penalty_threshold
        self.outlier_z = outlier_z_threshold
        logger.info(f"PeerNormalizer initialized (k={self.k}, z_outlier={self.outlier_z})")

    def normalize(
        self,
        convergence_by_policy_area: dict[str, float],
        peer_distributions: dict[str, dict[str, float]],
        baseline_confidence: float
    ) -> PeerNormalization:
        """
        Normalize scores against peer distributions

        Args:
            convergence_by_policy_area: Scores by policy area
            peer_distributions: Mean and std dev for each policy area
            baseline_confidence: Starting confidence level

        Returns:
            PeerNormalization with adjusted confidence
        """
        # Calculate z-scores
        z_scores = self._calculate_z_scores(
            convergence_by_policy_area,
            peer_distributions
        )

        # Identify outliers
        outlier_areas = self._identify_outliers(z_scores)

        # Count low-performing areas
        low_performers = [
            area for area, z in z_scores.items()
            if z < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1008_20", 1.0)
        ]

        # Adjust confidence
```

```
        adjusted_confidence = self._adjust_confidence(
            baseline_confidence,
            z_scores,
            low_performers
        )

        # Determine peer position
        peer_position = self._determine_position(z_scores)

        return PeerNormalization(
            z_scores=z_scores,
            adjusted_confidence=adjusted_confidence,
            peer_position=peer_position,
            outlier_areas=outlier_areas,
            metadata={
                "num_policy_areas": len(convergence_by_policy_area),
                "low_performers": len(low_performers),
                "penalty_threshold": self.k
            }
        )

    def _calculate_z_scores(
        self,
        convergence: dict[str, float],
        peer_distributions: dict[str, dict[str, float]]
    ) -> dict[str, float]:
        """Calculate z-scores for each policy area"""
        z_scores = {}

        for area, score in convergence.items():
            if area in peer_distributions:
                peer = peer_distributions[area]
                mean = peer.get("mean", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1044_40", 0.5))
                std = peer.get("std", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1045_38", 0.1))

                # Calculate z-score
                z = (score - mean) / std if std > 0 else ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1048_57", 0.0)

                z_scores[area] = z

        return z_scores

    def _identify_outliers(
        self,
        z_scores: dict[str, float]
    ) -> list[str]:
        """Identify outlier policy areas"""
        outliers = []

        for area, z in z_scores.items():
            if abs(z) > self.outlier_z:
                outliers.append(area)

        return outliers

```

```python
    def _adjust_confidence(
        self,
        baseline: float,
        z_scores: dict[str, float],
        low_performers: list[str]
    ) -> float:
        """Adjust confidence based on peer comparison"""
        adjusted = baseline

        # Penalize if too many low performers
        if len(low_performers) > self.k:
            penalty = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1078_22", 0.1) * (len(low_performers) - self.k)
            adjusted *= (ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1079_25", 1.0) - min(penalty, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1079_44", 0.5)))

        # Check if all within Â±ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1081_31", 0.5) z (tight distribution)
        all_tight = all(abs(z) <= ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1082_34", 0.5) for z in z_scores.values())

        if all_tight and len(z_scores) > 0:
            # Increase confidence for consistent performance
            adjusted *= 1.1

        return max(ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1088_19", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1088_28", 1.0), adjusted))

    def _determine_position(
        self,
        z_scores: dict[str, float]
    ) -> str:
        """Determine overall peer position"""
        if not z_scores:
            return "average"

        avg_z = statistics.mean(z_scores.values())

        if avg_z > ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1100_19", 0.5):
            return "above_average"
        elif avg_z < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1102_22", 0.5):
            return "below_average"
        else:
            return "average"

# ============================================================================
# MACRO PROMPTS FACADE
# ============================================================================

class MacroPromptsOrchestrator:
    """
    Orchestrator for all 5 macro-level analysis prompts

    Provides unified interface to execute all macro analyses
    """

```

```
    def __init__(self) -> None:
        """Initialize all macro prompt components"""
        self.coverage_stressor = CoverageGapStressor()
        self.contradiction_scanner = ContradictionScanner()
        self.portfolio_composer = BayesianPortfolioComposer()
        self.roadmap_optimizer = RoadmapOptimizer()
        self.peer_normalizer = PeerNormalizer()

        logger.info("MacroPromptsOrchestrator initialized with all 5 components")

    def execute_all(
        self,
        macro_data: dict[str, Any]
    ) -> dict[str, Any]:
        """
        Execute all 5 macro analyses

        Args:
            macro_data: Complete macro-level data including:
                - convergence_by_dimension
                - convergence_by_policy_area
                - missing_clusters
                - dimension_coverage
                - policy_area_coverage
                - micro_claims
                - meso_summary_signals
                - macro_narratives
                - meso_posteriors
                - cluster_weights
                - critical_gaps
                - dependency_graph
                - effort_estimates
                - impact_scores
                - peer_distributions
                - baseline_confidence

        Returns:
            Dict with results from all 5 analyses
        """
        results = {}

        # 1. Coverage & Structural Gap Analysis
        coverage_analysis = self.coverage_stressor.evaluate(
            convergence_by_dimension=macro_data.get("convergence_by_dimension", {}),
            missing_clusters=macro_data.get("missing_clusters", []),
            dimension_coverage=macro_data.get("dimension_coverage", {}),
            policy_area_coverage=macro_data.get("policy_area_coverage", {}),
            baseline_confidence=macro_data.get("baseline_confidence", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.MacroPromptsOrchestrator.__init__", "auto_param_L1165_70", 1.0))
        )
        results["coverage_analysis"] = asdict(coverage_analysis)

        # 2. Inter-Level Contradiction Scan
        contradiction_report = self.contradiction_scanner.scan(
            micro_claims=macro_data.get("micro_claims", []),
            meso_summary_signals=macro_data.get("meso_summary_signals", {}),
```

```
            macro_narratives=macro_data.get("macro_narratives", {})
        )
        results["contradiction_report"] = asdict(contradiction_report)

        # 3. Bayesian Portfolio Composition
        bayesian_portfolio = self.portfolio_composer.compose(
            meso_posteriors=macro_data.get("meso_posteriors", {}),
            cluster_weights=macro_data.get("cluster_weights", {}),
            reconciliation_penalties=macro_data.get("reconciliation_penalties")
        )
        results["bayesian_portfolio"] = asdict(bayesian_portfolio)

        # 4. Roadmap Optimization
        implementation_roadmap = self.roadmap_optimizer.optimize(
            critical_gaps=macro_data.get("critical_gaps", []),
            dependency_graph=macro_data.get("dependency_graph", {}),
            effort_estimates=macro_data.get("effort_estimates", {}),
            impact_scores=macro_data.get("impact_scores", {})
        )
        results["implementation_roadmap"] = asdict(implementation_roadmap)

        # 5. Peer Normalization
        peer_normalization = self.peer_normalizer.normalize(
            convergence_by_policy_area=macro_data.get("convergence_by_policy_area", {}),
            peer_distributions=macro_data.get("peer_distributions", {}),
            baseline_confidence=macro_data.get("baseline_confidence", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.MacroPromptsOrchestrator.__init__", "auto_param_L1198_70", 1.0))
        )
        results["peer_normalization"] = asdict(peer_normalization)

        logger.info("Completed all 5 macro analyses")
        return results



================================================================================
FILE: src/farfan_pipeline/analysis/meso_cluster_analysis.py
================================================================================

"""Meso-level analytics utilities for cluster evaluation prompts.

This module implements four independent helper functions that operationalise
the bespoke "Prompt Meso" specifications used by the analytics team:

* :func:`analyze_policy_dispersion` provides dispersion analytics, including
  coefficient of variation, gap analysis, and a light penalty framework.
* :func:`reconcile_cross_metrics` validates heterogeneous metric feeds against
  an authoritative macro reference and emits governance flags.
* :func:`compose_cluster_posterior` aggregates micro posteriors using a
  Bayesian-style roll-up while accounting for reconciliation penalties.
* :func:`calibrate_against_peers` situates the cluster against its peer group
  using inter-quartile comparisons and Tukey-style outlier detection.

The functions deliberately return both structured JSON-friendly payloads and a
short narrative string whenever the prompt mandates qualitative guidance.  The
implementation is dependency-light (standard library only) to keep it aligned
```

```
with the rest of the analytics toolbox.
"""

from __future__ import annotations

from dataclasses import dataclass
from functools import reduce
from statistics import fmean, pstdev
from typing import TYPE_CHECKING
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

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

def _to_float_sequence(values: Iterable[float]) -> list[float]:
    return [float(v) for v in values]

def _safe_mean(values: Iterable[float]) -> float:
    seq = _to_float_sequence(values)
    if not seq:
        return 0.0
    return float(fmean(seq))

def _safe_std(values: Iterable[float]) -> float:
    seq = _to_float_sequence(values)
    if len(seq) <= 1:
        return 0.0
    return float(pstdev(seq))

def _percentile(values: Sequence[float], percent: float) -> float:
    seq = sorted(_to_float_sequence(values))
    if not seq:
        return 0.0
    if percent <= 0:
        return seq[0]
    if percent >= 100:
        return seq[-1]
    k = (len(seq) - 1) * (percent / 100.0)
    lower_index = int(k)
    upper_index = min(lower_index + 1, len(seq) - 1)
    weight = k - lower_index
    return seq[lower_index] + weight * (seq[upper_index] - seq[lower_index])

def _gini(values: Iterable[float]) -> float:
    """Compute the Gini coefficient for a sequence of non-negative values."""

    seq = sorted(_to_float_sequence(values))
    if not seq:
        return 0.0
    if any(v < 0 for v in seq):
        raise ValueError("Gini coefficient is undefined for negative values")
    if all(v == 0 for v in seq):
        return 0.0
    total = sum(seq)
    if total == 0:
```

```
        # Non-negative numbers can only sum to zero if they are all zero, which is
        # handled above. Guard against floating point artefacts that leave a
        # near-zero denominator.
        return 0.0
    n = len(seq)
    weighted_sum = 0.0
    for i, value in enumerate(seq, start=1):
        weighted_sum += i * value
    gini = (2 * weighted_sum) / (n * total) - (n + 1) / n
    return float(gini)

def _tukey_bounds(p25: float, p75: float) -> tuple[float, float]:
    lower_quartile, upper_quartile = sorted((float(p25), float(p75)))
    iqr = upper_quartile - lower_quartile
    return (lower_quartile - 1.5 * iqr, upper_quartile + 1.5 * iqr)

def analyze_policy_dispersion(
    policy_area_scores: Mapping[str, float],
    peer_dispersion_stats: Mapping[str, float],
    thresholds: Mapping[str, float],
) -> tuple[dict[str, object], str]:
    """Evaluate intra-cluster dispersion and recommend a penalty.

    Parameters
    ----------
    policy_area_scores:
        Mapping of policy area names to their normalised scores.
    peer_dispersion_stats:
        Median dispersion statistics for comparable clusters. Expected keys are
        ``cv_median`` and ``gap_median``; missing keys are handled gracefully.
    thresholds:
        Warning/failure thresholds with keys ``cv_warn``, ``cv_fail``,
        ``gap_warn`` and ``gap_fail``.

    Returns
    -------
    Tuple[Dict[str, object], str]
        A tuple of the JSON-friendly payload and the five-to-six line narrative.
    """

    values = _to_float_sequence(policy_area_scores.values())
    mean_score = _safe_mean(values)
    std_score = _safe_std(values)
    cv = std_score / mean_score if mean_score else 0.0
    max_gap = float(max(values) - min(values)) if values else 0.0
    gini = _gini(values)

    peer_cv = float(peer_dispersion_stats.get("cv_median", cv))
    peer_gap = float(peer_dispersion_stats.get("gap_median", max_gap))

    cv_warn = float(thresholds.get("cv_warn", peer_cv))
    cv_fail = float(thresholds.get("cv_fail", peer_cv))
    gap_warn = float(thresholds.get("gap_warn", peer_gap))
    gap_fail = float(thresholds.get("gap_fail", peer_gap))

    severity = 0
```

```
    if cv > cv_warn or max_gap > gap_warn:
        severity = 1
    if cv > cv_fail or max_gap > gap_fail:
        severity = 2
    peer_escalation = cv > 1.5 * peer_cv or max_gap > 1.5 * peer_gap
    if peer_escalation or cv > 1.5 * cv_fail or max_gap > 1.5 * gap_fail:
        severity = 3

    classification = {
        0: "Concentrado",
        1: "Moderado",
        2: "Disperso",
        3: "CrÃ-tico",
    }[severity]

    penalty_components: list[float] = []
    if cv_fail:
        penalty_components.append(min(cv / cv_fail, 1.5))
    if gap_fail:
        penalty_components.append(min(max_gap / gap_fail, 1.5))
    peer_signal: list[float] = []
    if peer_cv:
        peer_signal.append(min(cv / peer_cv, 2.0))
    if peer_gap:
        peer_signal.append(min(max_gap / peer_gap, 2.0))

    base_penalty = _safe_mean(penalty_components) if penalty_components else 0.0
    peer_penalty = _safe_mean(peer_signal) if peer_signal else 0.0
    penalty = float(min(1.0, 0.6 * base_penalty + 0.4 * (peer_penalty - 1.0)))
    penalty = max(0.0, penalty)

    # Hypothetical normalisation of the lower tail: lift scores below Q1 to Q1.
    if values:
        q1 = float(_percentile(values, 25))
        normalised_values = [max(v, q1) for v in values]
        norm_mean = _safe_mean(normalised_values)
        norm_cv = _safe_std(normalised_values) / norm_mean if norm_mean else 0.0
        norm_gap = float(max(normalised_values) - min(normalised_values))
        mean_uplift = norm_mean - mean_score
    else:
        norm_cv = 0.0
        norm_gap = 0.0
        mean_uplift = 0.0

    json_payload = {
        "cv": cv,
        "max_gap": max_gap,
        "gini": gini,
        "class": classification,
        "penalty": penalty,
        "normalized_projection": {
            "adjusted_cv": norm_cv,
            "adjusted_max_gap": norm_gap,
            "mean_uplift": mean_uplift,
        },
    }
```

```

    lines = [
        f"La variabilidad intraclÃºster muestra un CV de {cv:.2f} frente al referente de {peer_cv:.2f}.",
        f"La brecha mÃ¡xima es de {max_gap:.1f} puntos, lo que sitÃºa la clasificaciÃ³n en nivel {classification}.",
        f"El coeficiente de Gini ({gini:.2f}) evidencia {’alta’ if gini > 0.3 else ’moderada’} concentraciÃ³n de resultados.",
        "La penalizaciÃ³n propuesta Ï\210 pondera desalineaciones internas y diferenciales contra los pares comparables.",
        "Si se normaliza la cola baja hacia el cuartil 25, el CV se reduce a "
        f"{norm_cv:.2f} y el gap a {norm_gap:.1f} con un uplift medio de {mean_uplift:.1f}.",
        "Persisten riesgos de sesgo de apreciaciÃ³n si se ignora la sensibilidad de la cola baja frente a shocks sectoriales.",
    ]
    narrative = "\n".join(lines[:6])

    return json_payload, narrative

@dataclass
class MetricViolation:
    metric_id: str
    unit_mismatch: bool = False
    stale_period: bool = False
    entity_misalignment: bool = False
    out_of_range: bool = False

    @calibrated_method("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict")
    def to_flag_dict(self) -> dict[str, object]:
        return {
            "metric_id": self.metric_id,
            "unit_mismatch": self.unit_mismatch,
            "stale_period": self.stale_period,
            "entity_misalignment": self.entity_misalignment,
            "out_of_range": self.out_of_range,
        }

def _convert_unit(
    value: float,
    from_unit: str,
    to_unit: str,
    crosswalk: Mapping[str, Mapping[str, float]],
) -> tuple[float, str]:
    if from_unit == to_unit:
        return value, to_unit
    conversions = crosswalk.get(from_unit, {})
    factor = conversions.get(to_unit)
    if factor is None:
        raise ValueError("Units are not convertible with provided crosswalk")
    return value * factor, to_unit

def reconcile_cross_metrics(
    aggregated_metrics: Iterable[Mapping[str, object]],
    macro_json: Mapping[str, object],
) -> dict[str, object]:
    """Validate heterogeneous metrics against an authoritative macro source."""

    reference: Mapping[str, Mapping[str, object]] = macro_json.get("metrics", {})  # type: ignore[assignment]
    crosswalk: Mapping[str, Mapping[str, float]] = macro_json.get("unit_crosswalk", {})  # type: ignore[assignment]

    validated_metrics: list[dict[str, object]] = []
```

```
    violations: list[dict[str, object]] = []

    for metric in aggregated_metrics:
        metric_id = str(metric.get("metric_id"))
        value = float(metric.get("value", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L246_42", 0.0)))
        unit = str(metric.get("unit")) if metric.get("unit") is not None else ""
        period = str(metric.get("period")) if metric.get("period") is not None else ""
        entity = str(metric.get("entity")) if metric.get("entity") is not None else ""

        expected = reference.get(metric_id, {})
        expected_unit = str(expected.get("unit", unit)) if expected else unit
        expected_period = str(expected.get("period", period)) if expected else period
        expected_entities = expected.get("entities") if isinstance(expected.get("entities"), list) else []
        lower_bound, upper_bound = expected.get("range", (None, None))

        violation = MetricViolation(metric_id)

        reconciled_value = value
        reconciled_unit = unit

        conversion_failed = False

        if expected_unit and unit and unit != expected_unit:
            try:
                reconciled_value, reconciled_unit = _convert_unit(value, unit, expected_unit, crosswalk)
            except ValueError:
                violation.unit_mismatch = True
                conversion_failed = True

        if expected_period and period and period != expected_period:
            violation.stale_period = True

        if expected_entities and entity and entity not in expected_entities:
            violation.entity_misalignment = True

        if not conversion_failed and lower_bound is not None and reconciled_value < float(lower_bound):
            violation.out_of_range = True
        if not conversion_failed and upper_bound is not None and reconciled_value > float(upper_bound):
            violation.out_of_range = True

        validated_metrics.append(
            {
                "metric_id": metric_id,
                "value": reconciled_value,
                "unit": reconciled_unit,
                "period": expected_period if expected_period else period,
                "entity": entity,
            }
        )

        if (
            violation.unit_mismatch
            or violation.stale_period
            or violation.entity_misalignment
            or violation.out_of_range
```

```
        ):
            violations.append(violation.to_flag_dict())

    total_checks = len(validated_metrics) * 4 if validated_metrics else 1
    total_violations = sum(
        violation[flag]
        for violation in violations
        for flag in ("unit_mismatch", "stale_period", "entity_misalignment", "out_of_range")
    )
    reconciled_confidence = max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L306_32", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L306_37", 1.0) - total_violations / total_checks)

    return {
        "metrics_validated": validated_metrics,
        "violations": violations,
        "reconciled_confidence": reconciled_confidence,
    }

def compose_cluster_posterior(
    micro_posteriors: Iterable[float],
    weighting_trace: Iterable[float] | None = None,
    reconciliation_penalties: Mapping[str, float] | None = None,
) -> tuple[dict[str, object], str]:
    """Combine micro posteriors and reconciliation penalties into a cluster view."""

    posts = _to_float_sequence(micro_posteriors)
    if not posts:
        raise ValueError("micro_posteriors cannot be empty")

    if weighting_trace is None:
        weights = [ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L326_19", 1.0)] * len(posts)
    else:
        weights = _to_float_sequence(weighting_trace)
        if len(weights) != len(posts):
            raise ValueError("weighting_trace must match micro_posteriors length")
        if any(w < 0 for w in weights):
            raise ValueError("weighting_trace values must be non-negative")
    if all(w == 0 for w in weights):
        weights = [ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L334_19", 1.0)] * len(posts)

    # Prevent degenerate/negative totals; fallback to uniform if needed.
    weights = [max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L337_19", 0.0), float(w)) for w in weights]
    total_weight = sum(weights)
    if total_weight == 0:
        raise ValueError("At least one weight must be positive")
    normalised_weights = [w / total_weight for w in weights]
    prior_meso = float(sum(p * w for p, w in zip(posts, normalised_weights, strict=True)))

    variance = float(sum(w * (p - prior_meso) ** 2 for p, w in zip(posts, normalised_weights, strict=True)))
    uncertainty_index = float(variance ** ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L345_42", 0.5))

    penalties_input = reconciliation_penalties or {}
    dispersion_penalty = float(penalties_input.get("dispersion_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L348_73", 0.0)))
```

```
    coverage_penalty = float(penalties_input.get("coverage_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L349_69", 0.0)))
    reconciliation_penalty = float(penalties_input.get("reconciliation_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L350_81", 0.0)))

    penalty_factor = reduce(
        lambda acc, val: acc * val,
        [
            max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L355_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L355_21", 1.0) - dispersion_penalty),
            max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L356_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L356_21", 1.0) - coverage_penalty),
            max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L357_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L357_21", 1.0) - reconciliation_penalty),
        ],
        ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L359_8", 1.0),
    )
    posterior_meso = float(prior_meso * penalty_factor)

    json_payload = {
        "prior_meso": prior_meso,
        "penalties": {
            "dispersion_penalty": dispersion_penalty,
            "coverage_penalty": coverage_penalty,
            "reconciliation_penalty": reconciliation_penalty,
        },
        "posterior_meso": posterior_meso,
        "uncertainty_index": uncertainty_index,
    }

    explanation_lines = [
        f"La media ponderada de las micro evidencias define un prior meso de {prior_meso:.3f}.",
        "Las penalizaciones por dispersiÃ³n, cobertura y reconciliaciÃ³n actÃºan de forma multiplicativa sobre el prior.",
        f"El ajuste integrado produce un posterior de {posterior_meso:.3f}, coherente con la gobernanza aplicada.",
        f"La incertidumbre residual (Ï\203 â\211\210 {uncertainty_index:.3f}) refleja la varianza remanente de las micro posteriors.",
    ]

    return json_payload, "\n".join(explanation_lines)

def calibrate_against_peers(
    policy_area_scores: Mapping[str, float],
    peer_context: Mapping[str, Mapping[str, float]],
) -> tuple[dict[str, object], str]:
    """Compare cluster scores against peer medians and inter-quartile ranges."""

    area_positions: dict[str, str] = {}
    outliers: dict[str, bool] = {}
    dispersion_values = _to_float_sequence(policy_area_scores.values())
    if dispersion_values:
        cluster_mean = _safe_mean(dispersion_values)
        cluster_std = _safe_std(dispersion_values)
        cluster_cv = cluster_std / cluster_mean if cluster_mean else ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L395_69", 0.0)
    else:
        cluster_cv = ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "cluster_cv", 0.0) # Refactored

```

```
    for area, score in policy_area_scores.items():
        peers = peer_context.get(area, {})
        median = float(peers.get("median", score))
        p25 = float(peers.get("p25", median))
        p75 = float(peers.get("p75", median))
        p75 - p25

        if score < p25:
            area_positions[area] = "below"
        elif score > p75:
            area_positions[area] = "above"
        else:
            area_positions[area] = "within"

        lower_bound, upper_bound = _tukey_bounds(p25, p75)
        outliers[area] = score < lower_bound or score > upper_bound

    json_payload = {
        "area_positions": area_positions,
        "outliers": outliers,
    }

    above_areas = [area for area, position in area_positions.items() if position == "above"]
    below_areas = [area for area, position in area_positions.items() if position == "below"]
    within_areas = [area for area, position in area_positions.items() if position == "within"]

    narrative_lines = [
        "El contraste con la mediana de los pares muestra un desempeÃ±o heterogÃ©neo por Ã¡rea." ,
        f"Se ubican por encima del IQR {', '.join(above_areas) if above_areas else 'ninguna Ã¡rea'}, mientras que {', '.join(below_areas) if below_areas else 'no hay caÃdas relevantes'} quedan por debajo.",
        f"Las Ã¡reas en zona intercuartÃlica ({', '.join(within_areas) if within_areas else 'sin registros'}) sostienen la base del clÃºster.",
        "Los outliers detectados mediante Tukey advierten focos crÃticos que requieren revisiÃ³n especÃfica.",
        f"Un municipio con media equiparable pero menor CV (~{cluster_cv:.2f}) ofrecerÃa narrativa mÃ¡s cohesionada, subrayando nuestra dispersiÃ³n relativa.",
        "Conviene integrar estos hallazgos en la calibraciÃ³n narrativa para evitar sobreponderar Ã©xitos aislados frente a rezagos estructurales.",
        "Recomendar explicitar cÃ³mo la dispersiÃ³n condiciona la comparabilidad con pares que exhiben mayor equilibrio interno.",
    ]

    return json_payload, "\n".join(narrative_lines[:7])




================================================================================
FILE: src/farfan_pipeline/analysis/micro_prompts.py
================================================================================

"""
Micro Prompts - Provenance Auditor, Bayesian Posterior Justification, and Anti-Milagro Stress Test
==================================================================================================

This module implements three critical micro-level analysis prompts:

1. PROVENANCE AUDITOR (QMCM Integrity Check):
   - Validates Questionâ\206\222Method Contribution Map consistency
   - Verifies provenance DAG integrity
```

```
   - Detects orphan nodes and schema mismatches
   - Monitors timing anomalies

2. BAYESIAN POSTERIOR JUSTIFICATION:
   - Explains signal contributions to posterior probability
   - Ranks signals by marginal impact
   - Identifies discarded signals
   - Justifies test types (Hoop, Smoking-Gun, etc.)

3. ANTI-MILAGRO STRESS TEST:
   - Detects structural fragility in causal chains
   - Evaluates proportionality pattern density
   - Simulates node removal to test robustness
   - Identifies non-proportional jumps

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

from __future__ import annotations

import logging
import time
import warnings
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from typing import Any

import numpy as np

# from farfan_pipeline import get_parameter_loader  # CALIBRATION DISABLED
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

# ============================================================================
# PROVENANCE AUDITOR - QMCM INTEGRITY CHECK
# ============================================================================

@dataclass
class QMCMRecord:
    """Record in the Questionâ\206\222Method Contribution Map

    Aligned with questionnaire_monolith.json structure:
    - base_slot: Question slot identifier from monolith
    - scoring_modality: Scoring mechanism (binary, ordinal, numeric, etc.)
    """
    question_id: str
    method_fqn: str
    contribution_weight: float
    timestamp: float
    output_schema: dict[str, Any]
    base_slot: str | None = field(default=None)  # From questionnaire monolith
    scoring_modality: str | None = field(default=None)  # From questionnaire monolith
    metadata: dict[str, Any] = field(default_factory=dict)
```

```

@dataclass
class ProvenanceNode:
    """Node in the provenance DAG"""
    node_id: str
    node_type: str  # ’input’, ’method’, ’output’
    parent_ids: list[str]
    qmcm_record_id: str | None = None
    timing: float = 0.0
    metadata: dict[str, Any] = field(default_factory=dict)

@dataclass
class ProvenanceDAG:
    """Provenance directed acyclic graph"""
    nodes: dict[str, ProvenanceNode]
    edges: list[tuple[str, str]]  # (from_node_id, to_node_id)

    @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceDAG.get_root_nodes")
    def get_root_nodes(self) -> list[str]:
        """Get nodes without parents (primary inputs)"""
        return [nid for nid, node in self.nodes.items() if not node.parent_ids]

    @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceDAG.get_orphan_nodes")
    def get_orphan_nodes(self) -> list[str]:
        """Get nodes without parents that are not primary inputs"""
        return [
            nid for nid, node in self.nodes.items()
            if not node.parent_ids and node.node_type != ’input’
        ]

@dataclass
class AuditResult:
    """Result of provenance audit"""
    missing_qmcm: list[str]  # Node IDs without QMCM records
    orphan_nodes: list[str]  # Nodes without proper parents
    schema_mismatches: list[dict[str, Any]]  # Schema violations
    latency_anomalies: list[dict[str, Any]]  # Timing outliers
    contribution_weights: dict[str, float]  # Method contribution distribution
    severity: str  # ’LOW’, ’MEDIUM’, ’HIGH’, ’CRITICAL’
    narrative: str  # 3-4 line explanation
    timestamp: float = field(default_factory=time.time)

class ProvenanceAuditor:
    """
    ROLE: Provenance Auditor [data governance]
    GOAL: Verify QMCM consistency and provenance DAG integrity

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(
```

```
        self,
        p95_latency_threshold: float | None = None,
        method_contracts: dict[str, dict[str, Any]] | None = None
    ) -> None:
        """
        Initialize provenance auditor

        Args:
            p95_latency_threshold: Historical p95 latency for anomaly detection
            method_contracts: Expected output schemas by method
        """
        warnings.warn(
            "ProvenanceAuditor is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.p95_threshold = p95_latency_threshold or 1000.0
        self.method_contracts = method_contracts or {}
        self.logger = logging.getLogger(self.__class__.__name__)

    def audit(
        self,
        micro_answer: Any,  # MicroLevelAnswer object
        evidence_registry: dict[str, QMCMRecord],
        provenance_dag: ProvenanceDAG,
        method_contracts: dict[str, dict[str, Any]] | None = None
    ) -> AuditResult:
        """
        Perform comprehensive provenance audit

        MANDATES:
        1. Validate 1:1 correspondence between DAG nodes and QMCM records
        2. Confirm no orphan nodes (except primary inputs)
        3. Check timing drift (flag if > p95 historical)
        4. Verify output_schema compliance
        5. Emit JSON audit + narrative

        Args:
            micro_answer: MicroLevelAnswer object to audit
            evidence_registry: QMCM records indexed by ID
            provenance_dag: Provenance DAG structure
            method_contracts: Expected schemas (optional override)

        Returns:
            AuditResult with findings and severity assessment
        """
        contracts = method_contracts or self.method_contracts

        # 1. Validate QMCM correspondence
        missing_qmcm = self._check_qmcm_correspondence(provenance_dag, evidence_registry)

        # 2. Detect orphan nodes
        orphan_nodes = provenance_dag.get_orphan_nodes()
```

```

        # 3. Check timing anomalies
        latency_anomalies = self._check_latency_anomalies(provenance_dag)

        # 4. Verify schema compliance
        schema_mismatches = self._check_schema_compliance(
            provenance_dag, evidence_registry, contracts
        )

        # 5. Calculate contribution weights
        contribution_weights = self._calculate_contribution_weights(evidence_registry)

        # Determine severity
        severity = self._assess_severity(
            missing_qmcm, orphan_nodes, schema_mismatches, latency_anomalies
        )

        # Generate narrative
        narrative = self._generate_narrative(
            len(missing_qmcm), len(orphan_nodes),
            len(schema_mismatches), len(latency_anomalies), severity
        )

        return AuditResult(
            missing_qmcm=missing_qmcm,
            orphan_nodes=orphan_nodes,
            schema_mismatches=schema_mismatches,
            latency_anomalies=latency_anomalies,
            contribution_weights=contribution_weights,
            severity=severity,
            narrative=narrative
        )

    def _check_qmcm_correspondence(
        self, dag: ProvenanceDAG, registry: dict[str, QMCMRecord]
    ) -> list[str]:
        """Check 1:1 node-to-QMCM correspondence"""
        missing = []
        for node_id, node in dag.nodes.items():
            if node.node_type == 'method':
                if not node.qmcm_record_id or node.qmcm_record_id not in registry:
                    missing.append(node_id)
        return missing

    @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor._check_latency_anomalies")
    def _check_latency_anomalies(self, dag: ProvenanceDAG) -> list[dict[str, Any]]:
        """Detect timing outliers beyond p95 threshold"""
        anomalies = []
        for node_id, node in dag.nodes.items():
            if node.timing > self.p95_threshold:
                anomalies.append({
                    'node_id': node_id,
                    'timing': node.timing,
                    'threshold': self.p95_threshold,
                    'excess': node.timing - self.p95_threshold
                })
```

```
        return anomalies

    def _check_schema_compliance(
        self,
        dag: ProvenanceDAG,
        registry: dict[str, QMCMRecord],
        contracts: dict[str, dict[str, Any]]
    ) -> list[dict[str, Any]]:
        """Verify method outputs match expected schemas"""
        mismatches = []
        for node_id, node in dag.nodes.items():
            if node.node_type == 'method' and node.qmcm_record_id:
                record = registry.get(node.qmcm_record_id)
                if record and record.method_fqn in contracts:
                    expected = contracts[record.method_fqn]
                    actual = record.output_schema

                    if not self._schemas_match(expected, actual):
                        mismatches.append({
                            'node_id': node_id,
                            'method': record.method_fqn,
                            'expected_schema': expected,
                            'actual_schema': actual
                        })
        return mismatches

    @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor._schemas_match")
    def _schemas_match(self, expected: dict[str, Any], actual: dict[str, Any]) -> bool:
        """Check if actual schema matches expected schema"""
        # Simple type-based matching
        return all(key in actual for key, expected_type in expected.items())

    def _calculate_contribution_weights(
        self, registry: dict[str, QMCMRecord]
    ) -> dict[str, float]:
        """Calculate method contribution distribution"""
        weights = defaultdict(float)
        for record in registry.values():
            weights[record.method_fqn] += record.contribution_weight
        return dict(weights)

    def _assess_severity(
        self,
        missing_qmcm: list[str],
        orphan_nodes: list[str],
        schema_mismatches: list[dict[str, Any]],
        latency_anomalies: list[dict[str, Any]]
    ) -> str:
        """Assess overall audit severity"""
        total_issues = (
            len(missing_qmcm) + len(orphan_nodes) +
            len(schema_mismatches) + len(latency_anomalies)
        )

        if total_issues == 0:
            return 'LOW'
```

```
        elif total_issues <= 2:
            return 'MEDIUM'
        elif total_issues <= 5:
            return 'HIGH'
        else:
            return 'CRITICAL'

    def _generate_narrative(
        self, missing: int, orphans: int, mismatches: int, anomalies: int, severity: str
    ) -> str:
        """Generate 3-4 line narrative summary"""
        narrative = f"Provenance audit completed with {severity} severity. "

        if missing > 0:
            narrative += f"Found {missing} nodes without QMCM records. "
        if orphans > 0:
            narrative += f"Detected {orphans} orphan nodes requiring parent linkage. "
        if mismatches > 0:
            narrative += f"Identified {mismatches} schema violations. "
        if anomalies > 0:
            narrative += f"Flagged {anomalies} latency anomalies exceeding p95. "

        if severity == 'LOW':
            narrative += "All critical integrity checks passed."
        elif severity == 'CRITICAL':
            narrative += "Immediate remediation required for data governance."

        return narrative

    @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor.to_json")
    def to_json(self, result: AuditResult) -> dict[str, Any]:
        """Export audit result as JSON"""
        return asdict(result)

# ============================================================================
# BAYESIAN POSTERIOR JUSTIFICATION
# ============================================================================

@dataclass
class Signal:
    """Signal contributing to posterior probability"""
    test_type: str  # 'Hoop', 'Smoking-Gun', 'Straw-in-Wind', 'Doubly-Decisive'
    likelihood: float
    weight: float
    raw_evidence_id: str
    reconciled: bool
    delta_posterior: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.ProvenanceAuditor.to_json", "auto_param_L318_29", 0.0)
    reason: str = ""

@dataclass
class PosteriorJustification:
    """Bayesian posterior justification result"""
    prior: float
    posterior: float
    signals_ranked: list[dict[str, Any]]  # Signals sorted by |Î\224|
    discarded_signals: list[dict[str, Any]]  # Signals rejected
```

```
    anti_miracle_cap_applied: bool
    cap_delta: float  # How much was capped
    robustness_narrative: str  # 5-6 line synthesis
    timestamp: float = field(default_factory=time.time)

class BayesianPosteriorExplainer:
    """
    ROLE: Probabilistic Explainer [causal inference]
    GOAL: Explain signal contributions to final posterior

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(self, anti_miracle_cap: float = 0.95) -> None:
        """
        Initialize Bayesian posterior explainer

        Args:
            anti_miracle_cap: Maximum posterior probability (anti-miracle constraint)
        """
        warnings.warn(
            "BayesianPosteriorExplainer is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.anti_miracle_cap = anti_miracle_cap
        self.logger = logging.getLogger(self.__class__.__name__)

    def explain(
        self,
        prior: float,
        signals: list[Signal],
        posterior: float
    ) -> PosteriorJustification:
        """
        Explain how each signal contributed to posterior

        MANDATES:
        1. Order signals by absolute marginal impact |Î\224|
        2. Mark discarded signals (contract violation or reconciliation failure)
        3. Justify test_type in 1 line each
        4. Explain anti-miracle cap application

        Args:
            prior: Initial probability
            signals: List of signals with test types and likelihoods
            posterior: Final posterior probability

```

```
        Returns:
            PosteriorJustification with ranked signals and narrative
        """
        # Rank signals by marginal impact
        signals_ranked = self._rank_signals_by_impact(signals)

        # Identify discarded signals
        discarded = [s for s in signals if not s.reconciled]

        # Check if anti-miracle cap was applied
        cap_applied = posterior > self.anti_miracle_cap
        cap_delta = max(0, posterior - self.anti_miracle_cap) if cap_applied else ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.BayesianPosterio
rExplainer.__init__", "auto_param_L380_82", 0.0)

        # Adjust posterior if capped
        final_posterior = min(posterior, self.anti_miracle_cap)

        # Generate robustness narrative
        narrative = self._generate_robustness_narrative(
            prior, final_posterior, signals_ranked, discarded, cap_applied, cap_delta
        )

        # Convert signals to dict format
        ranked_dicts = [self._signal_to_dict(s) for s in signals_ranked]
        discarded_dicts = [self._signal_to_dict(s) for s in discarded]

        return PosteriorJustification(
            prior=prior,
            posterior=final_posterior,
            signals_ranked=ranked_dicts,
            discarded_signals=discarded_dicts,
            anti_miracle_cap_applied=cap_applied,
            cap_delta=cap_delta,
            robustness_narrative=narrative
        )

    @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._rank_signals_by_impact")
    def _rank_signals_by_impact(self, signals: list[Signal]) -> list[Signal]:
        """Sort signals by absolute marginal impact"""
        # Only rank reconciled signals
        valid_signals = [s for s in signals if s.reconciled]

        # Sort by |delta_posterior| descending
        ranked = sorted(valid_signals, key=lambda s: abs(s.delta_posterior), reverse=True)

        # Add reasons based on test type
        for i, signal in enumerate(ranked):
            signal.reason = self._justify_test_type(signal.test_type, i + 1)

        return ranked

    @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._justify_test_type")
    def _justify_test_type(self, test_type: str, rank: int) -> str:
        """Generate 1-line justification for test type"""
        justifications = {
            'Hoop': f"Rank {rank}: Necessary condition test - failure eliminates hypothesis",
```

```
            'Smoking-Gun': f"Rank {rank}: Sufficient condition test - passage strongly confirms hypothesis",
            'Straw-in-Wind': f"Rank {rank}: Weak evidential test - provides marginal confirmation",
            'Doubly-Decisive': f"Rank {rank}: Necessary and sufficient - critical determining factor"
        }
        return justifications.get(test_type, f"Rank {rank}: {test_type} test applied")

    @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._signal_to_dict")
    def _signal_to_dict(self, signal: Signal) -> dict[str, Any]:
        """Convert Signal to dictionary"""
        return {
            'rank': 0,  # Will be set by caller if needed
            'test_type': signal.test_type,
            'delta_posterior': signal.delta_posterior,
            'kept': signal.reconciled,
            'reason': signal.reason,
            'likelihood': signal.likelihood,
            'weight': signal.weight,
            'evidence_id': signal.raw_evidence_id
        }

    def _generate_robustness_narrative(
        self,
        prior: float,
        posterior: float,
        signals: list[Signal],
        discarded: list[Signal],
        cap_applied: bool,
        cap_delta: float
    ) -> str:
        """Generate 5-6 line robustness synthesis"""
        narrative = f"Bayesian update from prior {prior:.3f} to posterior {posterior:.3f}. "

        if signals:
            top_signal = signals[0]
            narrative += f"Primary driver: {top_signal.test_type} test (Î\224={top_signal.delta_posterior:.3f}). "

        narrative += f"Integrated {len(signals)} reconciled signals. "

        if discarded:
            narrative += f"Discarded {len(discarded)} signals due to contract violations. "

        if cap_applied:
            narrative += f"Anti-miracle cap applied (Î\224={cap_delta:.3f} trimmed). "

        # Assess robustness
        if len(signals) >= 3 and not discarded:
            narrative += "High robustness with diverse evidential support."
        elif len(signals) >= 1:
            narrative += "Moderate robustness with limited triangulation."
        else:
            narrative += "Low robustness - insufficient evidential base."

        return narrative

    @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer.to_json")
    def to_json(self, result: PosteriorJustification) -> dict[str, Any]:
```

```
        """Export justification as JSON"""
        return asdict(result)

# ============================================================================
# ANTI-MILAGRO STRESS TEST
# ============================================================================

@dataclass
class CausalChain:
    """Causal chain of steps/edges"""
    steps: list[str]
    edges: list[tuple[str, str]]

    @calibrated_method("farfan_core.analysis.micro_prompts.CausalChain.length")
    def length(self) -> int:
        return len(self.steps)

@dataclass
class ProportionalityPattern:
    """Pattern indicating proportional causal relationship"""
    pattern_type: str  # 'linear', 'dose-response', 'threshold', 'mechanism'
    strength: float  # ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.CausalChain.length", "auto_param_L501_23", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.CausalChain.length", "auto_param_L501_27", 1.0)
    location: str  # Where in chain this appears

@dataclass
class StressTestResult:
    """Anti-milagro stress test result"""
    density: float  # Patterns per chain step
    simulated_drop: float  # Support score drop after node removal
    fragility_flag: bool  # True if drop > threshold
    explanation: str  # 3-line explanation
    pattern_coverage: float  # Fraction of chain covered by patterns
    missing_patterns: list[str]  # Required patterns not found
    timestamp: float = field(default_factory=time.time)

class AntiMilagroStressTester:
    """
    ROLE: Structural Stress Tester [causal integrity]
    GOAL: Detect dependence on non-proportional jumps

    DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
    Metadata:
        - creation_date: 2024-01-15
        - last_used_date: null
        - intended_phase: future_phase_7_9
        - deprecation_status: candidate_for_removal_2025Q2
    """

    def __init__(self, fragility_threshold: float = 0.3) -> None:
        """
        Initialize stress tester

        Args:
            fragility_threshold: Support score drop threshold for fragility
        """
```

```
        warnings.warn(
            "AntiMilagroStressTester is deprecated and scheduled for removal in 2025Q2. "
            "No usage detected as of February 2025. Intended for future_phase_7_9. "
            "Metadata: creation_date=2024-01-15, last_used_date=null, "
            "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
            DeprecationWarning,
            stacklevel=2
        )
        self.fragility_threshold = fragility_threshold
        self.logger = logging.getLogger(self.__class__.__name__)

    def stress_test(
        self,
        causal_chain: CausalChain,
        proportionality_patterns: list[ProportionalityPattern],
        missing_patterns: list[str]
    ) -> StressTestResult:
        """
        Stress test causal chain for structural fragility

        MANDATES:
        1. Evaluate pattern density vs chain length
        2. Simulate weak node removal and recalculate support
        3. Flag fragility if drop > Ï\204

        Args:
            causal_chain: Chain of causal steps
            proportionality_patterns: Detected proportionality patterns
            missing_patterns: Required patterns not found

        Returns:
            StressTestResult with fragility assessment
        """
        # 1. Calculate pattern density
        density = self._calculate_pattern_density(causal_chain, proportionality_patterns)

        # 2. Simulate node removal
        simulated_drop = self._simulate_node_removal(causal_chain, proportionality_patterns)

        # 3. Check fragility
        fragility_flag = simulated_drop > self.fragility_threshold

        # Calculate pattern coverage
        coverage = self._calculate_pattern_coverage(causal_chain, proportionality_patterns)

        # Generate explanation
        explanation = self._generate_explanation(density, simulated_drop, fragility_flag)

        return StressTestResult(
            density=density,
            simulated_drop=simulated_drop,
            fragility_flag=fragility_flag,
            explanation=explanation,
            pattern_coverage=coverage,
            missing_patterns=missing_patterns
        )
```

```

    def _calculate_pattern_density(
        self, chain: CausalChain, patterns: list[ProportionalityPattern]
    ) -> float:
        """Calculate patterns per chain step"""
        if chain.length() == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L582_19", 0.0)
        return len(patterns) / chain.length()

    def _calculate_pattern_coverage(
        self, chain: CausalChain, patterns: list[ProportionalityPattern]
    ) -> float:
        """Calculate fraction of chain covered by patterns"""
        if chain.length() == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L590_19", 0.0)

        # Count unique steps covered by patterns
        covered_steps = set()
        for pattern in patterns:
            # Extract step indices from pattern location
            # This is simplified - actual implementation would parse location
            covered_steps.add(pattern.location)

        return len(covered_steps) / chain.length()

    def _simulate_node_removal(
        self, chain: CausalChain, patterns: list[ProportionalityPattern]
    ) -> float:
        """Simulate removal of weak nodes and measure support drop"""
        if not patterns or chain.length() == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L606_19", 1.0)  # Maximum drop i
f no patterns

        # Calculate baseline support score
        baseline_support = self._calculate_support_score(patterns)

        # Identify weak patterns (bottom 25% by strength)
        if len(patterns) > 1:
            strengths = [p.strength for p in patterns]
            threshold = np.percentile(strengths, 25)
            strong_patterns = [p for p in patterns if p.strength > threshold]
        else:
            strong_patterns = patterns

        # Calculate support without weak patterns
        reduced_support = self._calculate_support_score(strong_patterns)

        # Calculate drop
        if baseline_support == 0:
            return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L624_19", 0.0)

        drop = (baseline_support - reduced_support) / baseline_support
        return max(ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L627_19", 0.0), min(ParameterLoa
derV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L627_28", 1.0), drop))  # Clamp to [0, 1]

    @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._calculate_support_score")
```

```
    def _calculate_support_score(self, patterns: list[ProportionalityPattern]) -> float:
        """Calculate overall support score from patterns"""
        if not patterns:
            return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._calculate_support_score", "auto_param_L633_19", 0.0)

        # Weighted average of pattern strengths
        total_weight = sum(p.strength for p in patterns)
        return total_weight / len(patterns)

    @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._generate_explanation")
    def _generate_explanation(self, density: float, drop: float, fragility: bool) -> str:
        """Generate 3-line explanation"""
        explanation = f"Pattern density: {density:.2f} patterns/step. "
        explanation += f"Simulated node removal causes {drop:.1%} support drop. "

        if fragility:
            explanation += "FRAGILITY DETECTED: Drop exceeds threshold, indicating structural weakness."
        else:
            explanation += "Robust structure: Support maintained under stress."

        return explanation

    @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json")
    def to_json(self, result: StressTestResult) -> dict[str, Any]:
        """Export stress test result as JSON"""
        return asdict(result)

# ============================================================================
# CONVENIENCE FUNCTIONS
# ============================================================================

def create_provenance_auditor(
    p95_latency: float | None = None,
    contracts: dict[str, dict[str, Any]] | None = None
) -> ProvenanceAuditor:
    """Factory function for ProvenanceAuditor"""
    return ProvenanceAuditor(p95_latency, contracts)

def create_posterior_explainer(anti_miracle_cap: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json", "auto_param_L668_57", 0.95)) -> BayesianPosteriorExplainer:
    """Factory function for BayesianPosteriorExplainer"""
    return BayesianPosteriorExplainer(anti_miracle_cap)

def create_stress_tester(fragility_threshold: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json", "auto_param_L672_54", 0.3)) -> AntiMilagroStressTester:
    """Factory function for AntiMilagroStressTester"""
    return AntiMilagroStressTester(fragility_threshold)



================================================================================
FILE: src/farfan_pipeline/analysis/recommendation_engine.py
================================================================================

# recommendation_engine.py - Rule-Based Recommendation Engine
"""
```

```
Recommendation Engine - Multi-Level Rule-Based Recommendations
================================================================

This module implements a rule-based recommendation engine that:
1. Loads and validates recommendation rules from JSON files
2. Evaluates conditions against score data at MICRO, MESO, and MACRO levels
3. Generates actionable recommendations with specific interventions
4. Renders templates with context-specific variable substitution

Supports three levels of recommendations:
- MICRO: Question-level recommendations (PA-DIM combinations)
- MESO: Cluster-level recommendations (CL01-CL04)
- MACRO: Plan-level strategic recommendations

Author: Integration Team
Version: 2.0.0
Python: 3.10+
"""

import logging
import re
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

import jsonschema
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

_REQUIRED_ENHANCED_FEATURES = {
    "template_parameterization",
    "execution_logic",
    "measurable_indicators",
    "unambiguous_time_horizons",
    "testable_verification",
    "cost_tracking",
    "authority_mapping",
}

# ============================================================================
# DATA STRUCTURES FOR RECOMMENDATIONS
# ============================================================================

@dataclass
class Recommendation:
    """
    Structured recommendation with full intervention details.

    Supports both v1.0 (simple) and v2.0 (enhanced with 7 advanced features):
    1. Template parameterization
    2. Execution logic
    3. Measurable indicators
    4. Unambiguous time horizons
```

```
    5. Testable verification
    6. Cost tracking
    7. Authority mapping
    """
    rule_id: str
    level: str  # MICRO, MESO, or MACRO
    problem: str
    intervention: str
    indicator: dict[str, Any]
    responsible: dict[str, Any]
    horizon: dict[str, Any]  # Changed from Dict[str, str] to support enhanced fields
    verification: list[Any]  # Changed from List[str] to support structured verification
    metadata: dict[str, Any] = field(default_factory=dict)

    # Enhanced fields (v2.0) - optional for backward compatibility
    execution: dict[str, Any] | None = None
    budget: dict[str, Any] | None = None
    template_id: str | None = None
    template_params: dict[str, Any] | None = None

    @calibrated_method("farfan_core.analysis.recommendation_engine.Recommendation.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for JSON serialization"""
        result = asdict(self)
        # Remove None values for cleaner output
        return {k: v for k, v in result.items() if v is not None}

@dataclass
class RecommendationSet:
    """
    Collection of recommendations with metadata
    """
    level: str
    recommendations: list[Recommendation]
    generated_at: str
    total_rules_evaluated: int
    rules_matched: int
    metadata: dict[str, Any] = field(default_factory=dict)

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationSet.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for JSON serialization"""
        return {
            'level': self.level,
            'recommendations': [r.to_dict() for r in self.recommendations],
            'generated_at': self.generated_at,
            'total_rules_evaluated': self.total_rules_evaluated,
            'rules_matched': self.rules_matched,
            'metadata': self.metadata
        }

# ============================================================================
# RECOMMENDATION ENGINE
# ============================================================================

class RecommendationEngine:
```

```
    """
    Core recommendation engine that evaluates rules and generates recommendations.

    Uses canonical notation for dimension and policy area validation.
    """

    def __init__(
        self,
        rules_path: str = "config/recommendation_rules_enhanced.json",
        schema_path: str = "rules/recommendation_rules.schema.json",
        questionnaire_provider=None,
        orchestrator=None
    ) -> None:
        """
        Initialize recommendation engine

        Args:
            rules_path: Path to recommendation rules JSON file
            schema_path: Path to JSON schema for validation
            questionnaire_provider: QuestionnaireResourceProvider instance (injected via DI)
            orchestrator: Orchestrator instance for accessing thresholds and patterns

        ARCHITECTURAL NOTE: Thresholds should come from questionnaire monolith
        via QuestionnaireResourceProvider, not from hardcoded values.
        """
        self.rules_path = Path(rules_path)
        self.schema_path = Path(schema_path)
        self.questionnaire_provider = questionnaire_provider
        self.orchestrator = orchestrator
        self.rules: dict[str, Any] = {}
        self.schema: dict[str, Any] = {}
        self.rules_by_level: dict[str, list[dict[str, Any]]] = {
            'MICRO': [],
            'MESO': [],
            'MACRO': []
        }

        # Load canonical notation for validation
        self._load_canonical_notation()

        # Load rules and schema
        self._load_schema()
        self._load_rules()

        logger.info(
            f"Recommendation engine initialized with "
            f"{len(self.rules_by_level['MICRO'])} MICRO, "
            f"{len(self.rules_by_level['MESO'])} MESO, "
            f"{len(self.rules_by_level['MACRO'])} MACRO rules"
        )

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_canonical_notation")
    def _load_canonical_notation(self) -> None:
        """Load canonical notation for validation"""
        try:
            from farfan_pipeline.core.canonical_notation import get_all_dimensions, get_all_policy_areas
```

```
            self.canonical_dimensions = get_all_dimensions()
            self.canonical_policy_areas = get_all_policy_areas()
            logger.info(
                f"Canonical notation loaded: {len(self.canonical_dimensions)} dimensions, "
                f"{len(self.canonical_policy_areas)} policy areas"
            )
        except Exception as e:
            logger.warning(f"Could not load canonical notation: {e}")
            self.canonical_dimensions = {}
            self.canonical_policy_areas = {}

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_schema")
    def _load_schema(self) -> None:
        """Load JSON schema for rule validation"""
        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import load_json

        try:
            self.schema = load_json(self.schema_path)
            logger.info(f"Loaded recommendation rules schema from {self.schema_path}")
        except Exception as e:
            logger.error(f"Failed to load schema: {e}")
            raise

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_rules")
    def _load_rules(self) -> None:
        """Load and validate recommendation rules"""
        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import load_json

        try:
            self.rules = load_json(self.rules_path)

            # Validate against schema
            jsonschema.validate(instance=self.rules, schema=self.schema)
            self._validate_ruleset_metadata()

            # Organize rules by level
            for rule in self.rules.get('rules', []):
                self._validate_rule(rule)
                level = rule.get('level')
                if level in self.rules_by_level:
                    self.rules_by_level[level].append(rule)

            logger.info(f"Loaded and validated {len(self.rules.get('rules', []))} rules from {self.rules_path}")
        except jsonschema.ValidationError as e:
            logger.error(f"Rule validation failed: {e.message}")
            raise
        except Exception as e:
            logger.error(f"Failed to load rules: {e}")
            raise

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine.reload_rules")
    def reload_rules(self) -> None:
        """Reload rules from disk (useful for hot-reloading)"""
        self.rules_by_level = {'MICRO': [], 'MESO': [], 'MACRO': []}
```

```
        self._load_rules()

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith")
    def get_thresholds_from_monolith(self) -> dict[str, Any]:
        """
        Get scoring thresholds from questionnaire monolith.

        Returns:
            Dictionary of thresholds by question_id or default thresholds

        ARCHITECTURAL NOTE: This method demonstrates proper access to
        questionnaire data via QuestionnaireResourceProvider, not direct I/O.
        """
        if self.questionnaire_provider is None:
            logger.warning("No questionnaire provider attached, using default thresholds")
            return {
                'default_micro_threshold': 2.0,
                'default_meso_threshold': 55.0,
                'default_macro_threshold': 65.0
            }

        # Get questionnaire data via provider
        questionnaire_data = self.questionnaire_provider.get_data()

        # Extract thresholds from monolith structure
        thresholds = {}
        blocks = questionnaire_data.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])

        for question in micro_questions:
            question_id = question.get('question_id')
            scoring_info = question.get('scoring', {})
            threshold = scoring_info.get('threshold')

            if question_id and threshold is not None:
                thresholds[question_id] = threshold

        logger.info(f"Loaded {len(thresholds)} thresholds from questionnaire monolith")
        return thresholds

    # ========================================================================
    # MICRO LEVEL RECOMMENDATIONS
    # ========================================================================

    def generate_micro_recommendations(
        self,
        scores: dict[str, float],
        context: dict[str, Any] | None = None
    ) -> RecommendationSet:
        """
        Generate MICRO-level recommendations based on PA-DIM scores

        Args:
            scores: Dictionary mapping "PA##-DIM##" to scores (ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_th
resholds_from_monolith", "auto_param_L279_63", 0.0)-3.0)
            context: Additional context for template rendering
```

```

        Returns:
            RecommendationSet with matched recommendations
        """
        recommendations = []
        rules_evaluated = 0

        for rule in self.rules_by_level['MICRO']:
            rules_evaluated += 1

            # Extract condition
            when = rule.get('when', {})
            pa_id = when.get('pa_id')
            dim_id = when.get('dim_id')
            score_lt = when.get('score_lt')

            # Build score key
            score_key = f"{pa_id}-{dim_id}"

            # Check if condition matches
            if score_key in scores and scores[score_key] < score_lt:
                # Render template
                template = rule.get('template', {})
                rendered = self._render_micro_template(template, pa_id, dim_id, context)

                # Create recommendation with enhanced fields (v2.0) if available
                rec = Recommendation(
                    rule_id=rule.get('rule_id'),
                    level='MICRO',
                    problem=rendered['problem'],
                    intervention=rendered['intervention'],
                    indicator=rendered['indicator'],
                    responsible=rendered['responsible'],
                    horizon=rendered['horizon'],
                    verification=rendered['verification'],
                    metadata={
                        'score_key': score_key,
                        'actual_score': scores[score_key],
                        'threshold': score_lt,
                        'gap': score_lt - scores[score_key]
                    },
                    # Enhanced fields (v2.0)
                    execution=rule.get('execution'),
                    budget=rule.get('budget'),
                    template_id=rendered.get('template_id'),
                    template_params=rendered.get('template_params')
                )
                recommendations.append(rec)

        return RecommendationSet(
            level='MICRO',
            recommendations=recommendations,
            generated_at=datetime.now(timezone.utc).isoformat(),
            total_rules_evaluated=rules_evaluated,
            rules_matched=len(recommendations)
        )
```

```

    def _render_micro_template(
        self,
        template: dict[str, Any],
        pa_id: str,
        dim_id: str,
        context: dict[str, Any] | None = None
    ) -> dict[str, Any]:
        """
        Render MICRO template with variable substitution

        Variables supported:
        - {{PAxx}}: Policy area (e.g., PA01)
        - {{DIMxx}}: Dimension (e.g., DIM01)
        - {{Q###}}: Question number (from context)
        """
        ctx = context or {}

        substitutions = {
            'PAxx': pa_id,
            'DIMxx': dim_id,
            'pa_id': pa_id,
            'dim_id': dim_id,
        }

        question_hint = ctx.get('question_id')
        template_params = template.get('template_params', {}) if isinstance(template, dict) else {}
        if isinstance(template_params, dict):
            for key, value in template_params.items():
                if isinstance(value, str):
                    substitutions.setdefault(key, value)
                    substitutions.setdefault(key.upper(), value)
                    if key == 'question_id':
                        question_hint = value

        if isinstance(question_hint, str):
            substitutions.setdefault(question_hint, question_hint)
            substitutions.setdefault('question_id', question_hint)
            substitutions.setdefault('Q001', question_hint)

        for key, value in ctx.items():
            if isinstance(value, str):
                substitutions.setdefault(key, value)

        return self._render_template(template, substitutions)

    # ========================================================================
    # MESO LEVEL RECOMMENDATIONS
    # ========================================================================

    def generate_meso_recommendations(
        self,
        cluster_data: dict[str, Any],
        context: dict[str, Any] | None = None
    ) -> RecommendationSet:
        """
```

```
        Generate MESO-level recommendations based on cluster performance

        Args:
            cluster_data: Dictionary with cluster metrics:
                {
                    'CL01': {'score': 75.0, 'variance': ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L398_56", 0.15), 'weak_pa': 'PA02'},
                    'CL02': {'score': 62.0, 'variance': ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L399_56", 0.22), 'weak_pa': 'PA05'},
                    ...
                }
            context: Additional context for template rendering

        Returns:
            RecommendationSet with matched recommendations
        """
        recommendations = []
        rules_evaluated = 0

        for rule in self.rules_by_level['MESO']:
            rules_evaluated += 1

            # Extract condition
            when = rule.get('when', {})
            cluster_id = when.get('cluster_id')
            score_band = when.get('score_band')
            variance_level = when.get('variance_level')
            variance_threshold = when.get('variance_threshold')
            weak_pa_id = when.get('weak_pa_id')

            # Get cluster data
            cluster = cluster_data.get(cluster_id, {})
            cluster_score = cluster.get('score', 0)
            cluster_variance = cluster.get('variance', 0)
            cluster_weak_pa = cluster.get('weak_pa')

            # Check conditions
            if not self._check_meso_conditions(
                cluster_score, cluster_variance, cluster_weak_pa,
                score_band, variance_level, variance_threshold, weak_pa_id
            ):
                continue

            # Render template
            template = rule.get('template', {})
            rendered = self._render_meso_template(template, cluster_id, context)

            # Create recommendation with enhanced fields (v2.0) if available
            rec = Recommendation(
                rule_id=rule.get('rule_id'),
                level='MESO',
                problem=rendered['problem'],
                intervention=rendered['intervention'],
                indicator=rendered['indicator'],
                responsible=rendered['responsible'],
                horizon=rendered['horizon'],
```

```
                verification=rendered['verification'],
                metadata={
                    'cluster_id': cluster_id,
                    'score': cluster_score,
                    'score_band': score_band,
                    'variance': cluster_variance,
                    'variance_level': variance_level,
                    'weak_pa': cluster_weak_pa
                },
                # Enhanced fields (v2.0)
                execution=rule.get('execution'),
                budget=rule.get('budget'),
                template_id=rendered.get('template_id'),
                template_params=rendered.get('template_params')
            )
            recommendations.append(rec)

        return RecommendationSet(
            level='MESO',
            recommendations=recommendations,
            generated_at=datetime.now(timezone.utc).isoformat(),
            total_rules_evaluated=rules_evaluated,
            rules_matched=len(recommendations)
        )

    def _check_meso_conditions(
        self,
        score: float,
        variance: float,
        weak_pa: str | None,
        score_band: str,
        variance_level: str,
        variance_threshold: float | None,
        weak_pa_id: str | None
    ) -> bool:
        """Check if MESO conditions are met"""
        # Check score band
        if score_band == 'BAJO' and score >= 55 or score_band == 'MEDIO' and (score < 55 or score >= 75) or score_band == 'ALTO' and score < 75:
            return False

        # Check variance level
        if variance_level == 'BAJA' and variance >= ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_52", 0.08) or variance_level == 'MEDIA' and (variance < ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_102", 0.08) or variance >= ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_122", 0.18)):
            return False
        elif variance_level == 'ALTA':
            if variance_threshold and variance < variance_threshold / 100 or not variance_threshold and variance < ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L491_115", 0.18):
                return False

        # Check weak PA if specified
        return not (weak_pa_id and weak_pa != weak_pa_id)

    def _render_meso_template(
        self,
```

```
        template: dict[str, Any],
        cluster_id: str,
        context: dict[str, Any] | None = None
    ) -> dict[str, Any]:
        """Render MESO template with variable substitution"""

        substitutions = {
            'cluster_id': cluster_id,
        }

        if isinstance(template, dict):
            params = template.get('template_params', {})
            if isinstance(params, dict):
                for key, value in params.items():
                    if isinstance(value, str):
                        substitutions.setdefault(key, value)
                        substitutions.setdefault(key.upper(), value)

        if context:
            for key, value in context.items():
                if isinstance(value, str):
                    substitutions.setdefault(key, value)

        return self._render_template(template, substitutions)

    # ========================================================================
    # MACRO LEVEL RECOMMENDATIONS
    # ========================================================================

    def generate_macro_recommendations(
        self,
        macro_data: dict[str, Any],
        context: dict[str, Any] | None = None
    ) -> RecommendationSet:
        """
        Generate MACRO-level strategic recommendations

        Args:
            macro_data: Dictionary with plan-level metrics:
                {
                    'macro_band': 'SATISFACTORIO',
                    'clusters_below_target': ['CL02', 'CL03'],
                    'variance_alert': 'MODERADA',
                    'priority_micro_gaps': ['PA01-DIM05', 'PA04-DIM04']
                }
            context: Additional context for template rendering

        Returns:
            RecommendationSet with matched recommendations
        """
        recommendations = []
        rules_evaluated = 0

        for rule in self.rules_by_level['MACRO']:
            rules_evaluated += 1

```

```
            # Extract condition
            when = rule.get('when', {})
            macro_band = when.get('macro_band')
            clusters_below = set(when.get('clusters_below_target', []))
            variance_alert = when.get('variance_alert')
            priority_gaps = set(when.get('priority_micro_gaps', []))

            # Get macro data
            actual_band = macro_data.get('macro_band')
            actual_clusters = set(macro_data.get('clusters_below_target', []))
            actual_variance = macro_data.get('variance_alert')
            actual_gaps = set(macro_data.get('priority_micro_gaps', []))

            # Check conditions
            if macro_band and macro_band != actual_band:
                continue
            if variance_alert and variance_alert != actual_variance:
                continue

            # Check if clusters match (subset or exact match)
            if clusters_below and not clusters_below.issubset(actual_clusters):
                # For MACRO, we want exact match or the rule's clusters to be present
                if clusters_below != actual_clusters and not actual_clusters.issubset(clusters_below):
                    continue

            # Check if priority gaps match (subset)
            if priority_gaps and not priority_gaps.issubset(actual_gaps):
                continue

            # Render template
            template = rule.get('template', {})
            rendered = self._render_macro_template(template, context)

            # Create recommendation with enhanced fields (v2.0) if available
            rec = Recommendation(
                rule_id=rule.get('rule_id'),
                level='MACRO',
                problem=rendered['problem'],
                intervention=rendered['intervention'],
                indicator=rendered['indicator'],
                responsible=rendered['responsible'],
                horizon=rendered['horizon'],
                verification=rendered['verification'],
                metadata={
                    'macro_band': actual_band,
                    'clusters_below_target': list(actual_clusters),
                    'variance_alert': actual_variance,
                    'priority_micro_gaps': list(actual_gaps)
                },
                # Enhanced fields (v2.0)
                execution=rule.get('execution'),
                budget=rule.get('budget'),
                template_id=rendered.get('template_id'),
                template_params=rendered.get('template_params')
            )
            recommendations.append(rec)
```

```
        return RecommendationSet(
            level='MACRO',
            recommendations=recommendations,
            generated_at=datetime.now(timezone.utc).isoformat(),
            total_rules_evaluated=rules_evaluated,
            rules_matched=len(recommendations)
        )

    def _render_macro_template(
        self,
        template: dict[str, Any],
        context: dict[str, Any] | None = None
    ) -> dict[str, Any]:
        """Render MACRO template with variable substitution"""

        substitutions = {}

        if context:
            for key, value in context.items():
                if isinstance(value, str):
                    substitutions.setdefault(key, value)

        if isinstance(template, dict):
            params = template.get('template_params', {})
            if isinstance(params, dict):
                for key, value in params.items():
                    if isinstance(value, str):
                        substitutions.setdefault(key, value)
                        substitutions.setdefault(key.upper(), value)

        return self._render_template(template, substitutions)

    # ========================================================================
    # UTILITY METHODS
    # ========================================================================

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._substitute_variables")
    def _substitute_variables(self, text: str, substitutions: dict[str, str]) -> str:
        """
        Substitute variables in text using {{variable}} syntax

        Args:
            text: Text with variables
            substitutions: Dictionary of variable_name -> value

        Returns:
            Text with variables substituted
        """
        result = text
        for var, value in substitutions.items():
            pattern = r'\{\{' + re.escape(var) + r'\}\}'
            result = re.sub(pattern, value, result)
        return result

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._render_template")
```

```
    def _render_template(self, template: dict[str, Any], substitutions: dict[str, str]) -> dict[str, Any]:
        """Recursively render a template applying substitutions to nested structures."""

        def render_value(value: Any) -> Any:
            if isinstance(value, str):
                return self._substitute_variables(value, substitutions)
            if isinstance(value, list):
                return [render_value(item) for item in value]
            if isinstance(value, dict):
                return {k: render_value(v) for k, v in value.items()}
            return value

        return render_value(template)

    # ========================================================================
    # VALIDATION UTILITIES
    # ========================================================================

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_rule")
    def _validate_rule(self, rule: dict[str, Any]) -> None:
        """Apply structural validation to guarantee rigorous recommendations."""
        rule_id = rule.get('rule_id')
        if not isinstance(rule_id, str) or not rule_id.strip():
            raise ValueError("Recommendation rule missing rule_id")

        level = rule.get('level')
        if level not in self.rules_by_level:
            raise ValueError(f"Rule {rule_id} declares unsupported level: {level}")

        when = rule.get('when', {})
        if not isinstance(when, dict):
            raise ValueError(f"Rule {rule_id} has invalid 'when' definition")

        if level == 'MICRO':
            self._validate_micro_when(rule_id, when)
        elif level == 'MESO':
            self._validate_meso_when(rule_id, when)
        elif level == 'MACRO':
            self._validate_macro_when(rule_id, when)

        template = rule.get('template')
        if not isinstance(template, dict):
            raise ValueError(f"Rule {rule_id} lacks a structured template")

        self._validate_template(rule_id, template, level)

        execution = rule.get('execution')
        if execution is None:
            raise ValueError(f"Rule {rule_id} is missing execution block required for enhanced rules")
        self._validate_execution(rule_id, execution)

        budget = rule.get('budget')
        if budget is None:
            raise ValueError(f"Rule {rule_id} is missing budget block required for enhanced rules")
        self._validate_budget(rule_id, budget)

```

```
    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_micro_when")
    def _validate_micro_when(self, rule_id: str, when: dict[str, Any]) -> None:
        required_keys = ('pa_id', 'dim_id', 'score_lt')
        for key in required_keys:
            if key not in when:
                raise ValueError(f"Rule {rule_id} missing '{key}' in MICRO condition")

        pa_id = when['pa_id']
        dim_id = when['dim_id']
        if not isinstance(pa_id, str) or not pa_id.strip():
            raise ValueError(f"Rule {rule_id} has invalid pa_id")
        if not isinstance(dim_id, str) or not dim_id.strip():
            raise ValueError(f"Rule {rule_id} has invalid dim_id")

        score_lt = when['score_lt']
        if not self._is_number(score_lt):
            raise ValueError(f"Rule {rule_id} has non-numeric MICRO threshold")
        if not 0 <= float(score_lt) <= 3:
            raise ValueError(f"Rule {rule_id} MICRO threshold must be between 0 and 3")

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_meso_when")
    def _validate_meso_when(self, rule_id: str, when: dict[str, Any]) -> None:
        cluster_id = when.get('cluster_id')
        if not isinstance(cluster_id, str) or not cluster_id.strip():
            raise ValueError(f"Rule {rule_id} missing cluster_id for MESO condition")

        condition_counter = 0

        score_band = when.get('score_band')
        if score_band is not None:
            if score_band not in {'BAJO', 'MEDIO', 'ALTO'}:
                raise ValueError(f"Rule {rule_id} has invalid MESO score_band")
            condition_counter += 1

        variance_level = when.get('variance_level')
        if variance_level is not None:
            if variance_level not in {'BAJA', 'MEDIA', 'ALTA'}:
                raise ValueError(f"Rule {rule_id} has invalid MESO variance_level")
            condition_counter += 1

        variance_threshold = when.get('variance_threshold')
        if variance_threshold is not None and not self._is_number(variance_threshold):
            raise ValueError(f"Rule {rule_id} has non-numeric variance_threshold")

        weak_pa_id = when.get('weak_pa_id')
        if weak_pa_id is not None:
            if not isinstance(weak_pa_id, str) or not weak_pa_id.strip():
                raise ValueError(f"Rule {rule_id} has invalid weak_pa_id")
            condition_counter += 1

        if condition_counter == 0:
            raise ValueError(
                f"Rule {rule_id} must specify at least one discriminant condition for MESO"
            )

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_macro_when")
```

```
    def _validate_macro_when(self, rule_id: str, when: dict[str, Any]) -> None:
        discriminants = 0

        macro_band = when.get('macro_band')
        if macro_band is not None:
            if not isinstance(macro_band, str) or not macro_band.strip():
                raise ValueError(f"Rule {rule_id} has invalid macro_band")
            discriminants += 1

        clusters = when.get('clusters_below_target')
        if clusters is not None:
            if not isinstance(clusters, list) or not clusters:
                raise ValueError(f"Rule {rule_id} must declare non-empty clusters_below_target")
            if not all(isinstance(item, str) and item.strip() for item in clusters):
                raise ValueError(f"Rule {rule_id} has invalid cluster identifiers")
            discriminants += 1

        variance_alert = when.get('variance_alert')
        if variance_alert is not None:
            if not isinstance(variance_alert, str) or not variance_alert.strip():
                raise ValueError(f"Rule {rule_id} has invalid variance_alert")
            discriminants += 1

        priority_gaps = when.get('priority_micro_gaps')
        if priority_gaps is not None:
            if not isinstance(priority_gaps, list) or not priority_gaps:
                raise ValueError(f"Rule {rule_id} must declare non-empty priority_micro_gaps")
            if not all(isinstance(item, str) and item.strip() for item in priority_gaps):
                raise ValueError(f"Rule {rule_id} has invalid priority_micro_gaps entries")
            discriminants += 1

        if discriminants == 0:
            raise ValueError(
                f"Rule {rule_id} must specify at least one MACRO discriminant condition"
            )

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_template")
    def _validate_template(self, rule_id: str, template: dict[str, Any], level: str) -> None:
        required_fields = ['problem', 'intervention', 'indicator', 'responsible', 'horizon', 'verification', 'template_id', 'template_params']
        for field in required_fields:
            if field not in template:
                raise ValueError(f"Rule {rule_id} template missing '{field}'")

        for text_field in ('problem', 'intervention'):
            value = template[text_field]
            if not isinstance(value, str):
                raise ValueError(f"Rule {rule_id} template field '{text_field}' must be text")
            stripped = value.strip()
            if len(stripped) < 40 or len(stripped.split()) < 12:
                raise ValueError(
                    f"Rule {rule_id} template field '{text_field}' lacks actionable detail"
                )

        indicator = template['indicator']
        if not isinstance(indicator, dict):
            raise ValueError(f"Rule {rule_id} indicator must be an object")
```

```
        for key in ('name', 'target', 'unit'):
            if key not in indicator:
                raise ValueError(f"Rule {rule_id} indicator missing '{key}' field")

        if not isinstance(indicator['name'], str) or len(indicator['name'].strip()) < 5:
            raise ValueError(f"Rule {rule_id} indicator name too short")

        target = indicator['target']
        if not self._is_number(target):
            raise ValueError(f"Rule {rule_id} indicator target must be numeric")

        unit = indicator['unit']
        if not isinstance(unit, str) or not unit.strip():
            raise ValueError(f"Rule {rule_id} indicator unit missing or empty")

        acceptable_range = indicator.get('acceptable_range')
        if acceptable_range is not None:
            if not isinstance(acceptable_range, list) or len(acceptable_range) != 2:
                raise ValueError(f"Rule {rule_id} acceptable_range must have two numeric bounds")
            if not all(self._is_number(bound) for bound in acceptable_range):
                raise ValueError(f"Rule {rule_id} acceptable_range values must be numeric")
            lower, upper = acceptable_range
            if float(lower) >= float(upper):
                raise ValueError(f"Rule {rule_id} acceptable_range lower bound must be < upper bound")

        template_id = template['template_id']
        if not isinstance(template_id, str) or not template_id.strip():
            raise ValueError(f"Rule {rule_id} template_id must be a non-empty string")

        template_params = template['template_params']
        if not isinstance(template_params, dict):
            raise ValueError(f"Rule {rule_id} template_params must be an object")
        allowed_param_keys = {'pa_id', 'dim_id', 'cluster_id', 'question_id'}
        unknown_params = set(template_params) - allowed_param_keys
        if unknown_params:
            raise ValueError(f"Rule {rule_id} template_params contains unsupported keys: {sorted(unknown_params)}")

        required_params: set[str] = set()
        if level == 'MICRO':
            required_params = {'pa_id', 'dim_id', 'question_id'}
        elif level == 'MESO':
            required_params = {'cluster_id'}

        missing_params = required_params - set(template_params)
        if missing_params:
            raise ValueError(
                f"Rule {rule_id} template_params missing required keys for {level}: {sorted(missing_params)}"
            )

        if level != 'MACRO' and not template_params:
            raise ValueError(f"Rule {rule_id} template_params cannot be empty for {level} level")

        responsible = template['responsible']
        if not isinstance(responsible, dict):
            raise ValueError(f"Rule {rule_id} responsible must be an object")
        for key in ('entity', 'role'):
```

```
            value = responsible.get(key)
            if not isinstance(value, str) or not value.strip():
                raise ValueError(f"Rule {rule_id} responsible missing '{key}'")

        partners = responsible.get('partners')
        if partners is None or not isinstance(partners, list) or not partners:
            raise ValueError(f"Rule {rule_id} responsible must enumerate partners")
        if any(not isinstance(partner, str) or not partner.strip() for partner in partners):
            raise ValueError(f"Rule {rule_id} responsible partners must be non-empty strings")

        horizon = template['horizon']
        if not isinstance(horizon, dict):
            raise ValueError(f"Rule {rule_id} horizon must be an object")
        for key in ('start', 'end'):
            value = horizon.get(key)
            if not isinstance(value, str) or not value.strip():
                raise ValueError(f"Rule {rule_id} horizon missing '{key}'")

        verification = template['verification']
        if not isinstance(verification, list) or not verification:
            raise ValueError(f"Rule {rule_id} must define verification artifacts")
        for artifact in verification:
            if not isinstance(artifact, dict):
                raise ValueError(
                    f"Rule {rule_id} verification entries must be structured dictionaries"
                )
            required_artifact_fields = (
                'id',
                'type',
                'artifact',
                'format',
                'approval_required',
                'approver',
                'due_date',
                'required_sections',
                'automated_check',
            )
            for key in required_artifact_fields:
                if key not in artifact:
                    raise ValueError(
                        f"Rule {rule_id} verification artifact missing required field '{key}'"
                    )
                # Special handling for boolean fields - they can be False
                if key in ('approval_required', 'automated_check'):
                    if not isinstance(artifact[key], bool):
                        raise ValueError(
                            f"Rule {rule_id} verification artifact field '{key}' must be a boolean"
                        )
                # Special handling for required_sections - must be a list
                elif key == 'required_sections':
                    if not isinstance(artifact[key], list) or not all(isinstance(s, str) and s.strip() for s in artifact[key]):
                        raise ValueError(
                            f"Rule {rule_id} verification required_sections must be a list of strings (may be empty)"
                        )
                # For other non-boolean fields, check for empty values
                elif not artifact[key]:
```

```
                raise ValueError(
                    f"Rule {rule_id} verification artifact field '{key}' cannot be empty"
                )

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_execution")
    def _validate_execution(self, rule_id: str, execution: dict[str, Any]) -> None:
        if not isinstance(execution, dict):
            raise ValueError(f"Rule {rule_id} execution block must be an object")

        required_keys = {
            'trigger_condition',
            'blocking',
            'auto_apply',
            'requires_approval',
            'approval_roles',
        }
        missing = required_keys - execution.keys()
        if missing:
            raise ValueError(f"Rule {rule_id} execution block missing keys: {sorted(missing)}")

        if not isinstance(execution['trigger_condition'], str) or not execution['trigger_condition'].strip():
            raise ValueError(f"Rule {rule_id} execution trigger_condition must be a non-empty string")
        for flag in ('blocking', 'auto_apply', 'requires_approval'):
            if not isinstance(execution[flag], bool):
                raise ValueError(f"Rule {rule_id} execution field '{flag}' must be boolean")

        roles = execution['approval_roles']
        if not isinstance(roles, list) or not roles:
            raise ValueError(f"Rule {rule_id} execution approval_roles must be a non-empty list")
        if any(not isinstance(role, str) or not role.strip() for role in roles):
            raise ValueError(f"Rule {rule_id} execution approval_roles must contain non-empty strings")

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_budget")
    def _validate_budget(self, rule_id: str, budget: dict[str, Any]) -> None:
        if not isinstance(budget, dict):
            raise ValueError(f"Rule {rule_id} budget block must be an object")

        required_keys = {'estimated_cost_cop', 'cost_breakdown', 'funding_sources', 'fiscal_year'}
        missing = required_keys - budget.keys()
        if missing:
            raise ValueError(f"Rule {rule_id} budget block missing keys: {sorted(missing)}")

        if not self._is_number(budget['estimated_cost_cop']):
            raise ValueError(f"Rule {rule_id} budget estimated_cost_cop must be numeric")

        cost_breakdown = budget['cost_breakdown']
        if not isinstance(cost_breakdown, dict) or not cost_breakdown:
            raise ValueError(f"Rule {rule_id} cost_breakdown must be a non-empty object")
        for key, value in cost_breakdown.items():
            if not isinstance(key, str) or not key.strip():
                raise ValueError(f"Rule {rule_id} cost_breakdown keys must be non-empty strings")
            if not self._is_number(value):
                raise ValueError(f"Rule {rule_id} cost_breakdown values must be numeric")

        funding_sources = budget['funding_sources']
        if not isinstance(funding_sources, list) or not funding_sources:
```

```
            raise ValueError(f"Rule {rule_id} funding_sources must be a non-empty list")
        for source in funding_sources:
            if not isinstance(source, dict):
                raise ValueError(f"Rule {rule_id} funding source entries must be objects")
            for key in ('source', 'amount', 'confirmed'):
                if key not in source:
                    raise ValueError(f"Rule {rule_id} funding source missing '{key}'")
            if not isinstance(source['source'], str) or not source['source'].strip():
                raise ValueError(f"Rule {rule_id} funding source name must be a non-empty string")
            if not self._is_number(source['amount']):
                raise ValueError(f"Rule {rule_id} funding source amount must be numeric")
            if not isinstance(source['confirmed'], bool):
                raise ValueError(f"Rule {rule_id} funding source confirmed flag must be boolean")

        fiscal_year = budget['fiscal_year']
        if not isinstance(fiscal_year, int):
            raise ValueError(f"Rule {rule_id} fiscal_year must be an integer")

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_ruleset_metadata")
    def _validate_ruleset_metadata(self) -> None:
        version = self.rules.get('version')
        if not isinstance(version, str) or not version.startswith('2.0'):
            raise ValueError(
                "Enhanced recommendation engine requires ruleset version 2.0"
            )

        features = self.rules.get('enhanced_features')
        if not isinstance(features, list) or not features:
            raise ValueError("Enhanced recommendation engine requires enhanced_features list")

        feature_set = {feature for feature in features if isinstance(feature, str)}
        missing = _REQUIRED_ENHANCED_FEATURES - feature_set
        if missing:
            raise ValueError(
                f"Enhanced recommendation rules missing required features: {sorted(missing)}"
            )

    @staticmethod
    def _is_number(value: Any) -> bool:
        return isinstance(value, (int, float)) and not isinstance(value, bool)

    def generate_all_recommendations(
        self,
        micro_scores: dict[str, float],
        cluster_data: dict[str, Any],
        macro_data: dict[str, Any],
        context: dict[str, Any] | None = None
    ) -> dict[str, RecommendationSet]:
        """
        Generate recommendations at all three levels

        Args:
            micro_scores: PA-DIM scores for MICRO recommendations
            cluster_data: Cluster metrics for MESO recommendations
            macro_data: Plan-level metrics for MACRO recommendations
            context: Additional context
```

```

        Returns:
            Dictionary with 'MICRO', 'MESO', and 'MACRO' recommendation sets
        """
        return {
            'MICRO': self.generate_micro_recommendations(micro_scores, context),
            'MESO': self.generate_meso_recommendations(cluster_data, context),
            'MACRO': self.generate_macro_recommendations(macro_data, context)
        }

    def export_recommendations(
        self,
        recommendations: dict[str, RecommendationSet],
        output_path: str,
        format: str = 'json'
    ) -> None:
        """
        Export recommendations to file

        Args:
            recommendations: Dictionary of recommendation sets
            output_path: Path to output file
            format: Output format ('json' or 'markdown')
        """
        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import save_json, write_text_file

        if format == 'json':
            save_json(
                {level: rec_set.to_dict() for level, rec_set in recommendations.items()},
                output_path
            )
        elif format == 'markdown':
            write_text_file(
                self._format_as_markdown(recommendations),
                output_path
            )
        else:
            raise ValueError(f"Unsupported format: {format}")

        logger.info(f"Exported recommendations to {output_path} in {format} format")

    @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._format_as_markdown")
    def _format_as_markdown(self, recommendations: dict[str, RecommendationSet]) -> str:
        """Format recommendations as Markdown"""
        lines = ["# Recomendaciones del Plan de Desarrollo\n"]

        for level in ['MICRO', 'MESO', 'MACRO']:
            rec_set = recommendations.get(level)
            if not rec_set:
                continue

            lines.append(f"\n## Nivel {level}\n")
            lines.append(f"**Generado:** {rec_set.generated_at}\n")
            lines.append(f"**Reglas evaluadas:** {rec_set.total_rules_evaluated}\n")
            lines.append(f"**Recomendaciones:** {rec_set.rules_matched}\n")
```

```

            for i, rec in enumerate(rec_set.recommendations, 1):
                lines.append(f"\n### {i}. {rec.rule_id}\n")
                lines.append(f"**Problema:** {rec.problem}\n")
                lines.append(f"\n**IntervenciÃ³n:** {rec.intervention}\n")
                lines.append("\n**Indicador:**")
                lines.append(f"- Nombre: {rec.indicator.get('name')}")
                lines.append(f"- Meta: {rec.indicator.get('target')} {rec.indicator.get('unit')}\n")
                lines.append(f"\n**Responsable:** {rec.responsible.get('entity')} ({rec.responsible.get('role')})\n")
                lines.append(f"**Socios:** {', '.join(rec.responsible.get('partners', []))}\n")
                lines.append(f"\n**Horizonte:** {rec.horizon.get('start')} â\206\222 {rec.horizon.get('end')}\n")
                lines.append("\n**VerificaciÃ³n:**")
                for v in rec.verification:
                    if isinstance(v, dict):
                        descriptor = f"[{v.get('type', 'ARTIFACT')}] {v.get('artifact', 'Sin artefacto')}"
                        due = v.get('due_date')
                        approver = v.get('approver')
                        suffix_parts: list[str] = []
                        if due:
                            suffix_parts.append(f"entrega: {due}")
                        if approver:
                            suffix_parts.append(f"aprueba: {approver}")
                        suffix = f" ({'; '.join(suffix_parts)})" if suffix_parts else ""
                        lines.append(f"- {descriptor}{suffix}")
                        sections = v.get('required_sections') or []
                        if sections:
                            lines.append(
                                "  - Secciones requeridas: " + ", ".join(str(section) for section in sections)
                            )
                    else:
                        lines.append(f"- {v}")
                lines.append("")

        return "\n".join(lines)

# ============================================================================
# CONVENIENCE FUNCTIONS
# ============================================================================

def load_recommendation_engine(
    rules_path: str = "config/recommendation_rules_enhanced.json",
    schema_path: str = "rules/recommendation_rules.schema.json"
) -> RecommendationEngine:
    """
    Convenience function to load recommendation engine

    Args:
        rules_path: Path to rules JSON
        schema_path: Path to schema JSON

    Returns:
        Initialized RecommendationEngine
    """
    return RecommendationEngine(rules_path=rules_path, schema_path=schema_path)

# Note: Main entry point removed to maintain I/O boundary separation.
```

```
# For usage examples, see examples/ directory.



================================================================================
FILE: src/farfan_pipeline/analysis/report_assembly.py
================================================================================

"""
Report Assembly Module - Production Grade v2.0
================================================

This module assembles comprehensive policy analysis reports by:
1. Loading questionnaire monolith via factory (I/O boundary)
2. Accessing patterns via QuestionnaireResourceProvider (single source of truth)
3. Integrating with evidence registry and QMCM hooks
4. Producing structured, traceable reports with cryptographic verification

Architectural Compliance:
- REQUIREMENT 1: Uses QuestionnaireResourceProvider for pattern extraction
- REQUIREMENT 2: All I/O via factory.py
- REQUIREMENT 3: Receives dependencies via dependency injection
- REQUIREMENT 4: Domain-specific exceptions with structured payloads
- REQUIREMENT 5: Pydantic contracts for data validation
- REQUIREMENT 6: Cryptographic verification (SHA-256)
- REQUIREMENT 7: Structured JSON logging
- REQUIREMENT 8: Parameter externalization via calibration system

Author: Integration Team
Version: 2.0.0
Python: 3.10+
"""

from __future__ import annotations

import hashlib
import json
import logging
import uuid
from datetime import datetime, timezone
from typing import TYPE_CHECKING, Any

from pydantic import BaseModel, ConfigDict, Field, field_validator
# Calibration parameters - loaded at runtime if calibration system available
try:
    from farfan_pipeline.core.parameters import ParameterLoaderV2
except (ImportError, AttributeError):
    # Fallback: use explicit defaults if calibration system not available
    _PARAM_LOADER = None

# Calibrated method decorator stub (calibration system not available)
def calibrated_method(method_name: str):
    """No-op decorator stub for compatibility when calibration system unavailable."""
    def decorator(func):
        return func
    return decorator
```

```

if TYPE_CHECKING:
    from pathlib import Path

logger = logging.getLogger(__name__)

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

class ReportAssemblyException(Exception):
    """Base exception for report assembly operations with structured payloads."""

    def __init__(
        self,
        message: str,
        details: dict[str, Any] | None = None,
        stage: str | None = None,
        recoverable: bool = False,
        event_id: str | None = None
    ) -> None:
        self.message = message
        self.details = details or {}
        self.stage = stage
        self.recoverable = recoverable
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(self._format_message())

    def _format_message(self) -> str:
        """Format error message with structured information."""
        parts = ["[ReportAssembly Error]"]
        if self.stage:
            parts.append(f"[Stage: {self.stage}]")
        parts.append(f"[EventID: {self.event_id[:8]}]")
        parts.append(self.message)
        if self.details:
            parts.append(f"Details: {json.dumps(self.details, indent=2)}")
        return " ".join(parts)

    def to_dict(self) -> dict[str, Any]:
        """Convert exception to structured dictionary."""
        return {
            'error_type': self.__class__.__name__,
            'message': self.message,
            'details': self.details,
            'stage': self.stage,
            'recoverable': self.recoverable,
            'event_id': self.event_id
        }


class ReportValidationError(ReportAssemblyException):
    """Raised when report data validation fails."""
    pass


```

```
class ReportIntegrityError(ReportAssemblyException):
    """Raised when cryptographic verification fails (hash mismatch)."""
    pass


class ReportExportError(ReportAssemblyException):
    """Raised when report export to file fails."""
    pass


# ============================================================================
# UTILITY FUNCTIONS
# ============================================================================

def compute_content_digest(content: str | bytes | dict[str, Any]) -> str:
    """
    Compute SHA-256 digest of content in a deterministic way.

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

    Returns:
        Hexadecimal SHA-256 digest (64 characters)

    Raises:
        ReportValidationError: If content type is unsupported
    """
    if isinstance(content, dict):
        # Sort keys for deterministic JSON
        content_str = json.dumps(content, sort_keys=True, ensure_ascii=True, separators=(',', ':'))
        content_bytes = content_str.encode('utf-8')
    elif isinstance(content, str):
        content_bytes = content.encode('utf-8')
    elif isinstance(content, bytes):
        content_bytes = content
    else:
        raise ReportValidationError(
            f"Cannot compute digest for type {type(content).__name__}",
            details={'content_type': type(content).__name__},
            stage="digest_computation"
        )

    return hashlib.sha256(content_bytes).hexdigest()


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

    Returns:
        ISO-8601 timestamp string (UTC timezone)
    """
    return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')


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

```
# PYDANTIC CONTRACT MODELS
# ============================================================================

class ReportMetadata(BaseModel):
    """Enhanced metadata for analysis report with cryptographic traceability."""

    model_config = ConfigDict(
        frozen=True,
        extra='forbid',
        validate_assignment=True,
        str_strip_whitespace=True,
    )

    report_id: str = Field(..., description="Unique report identifier", min_length=1)
    generated_at: str = Field(
        default_factory=utc_now_iso,
        description="UTC timestamp in ISO-8601 format"
    )
    monolith_version: str = Field(..., description="Questionnaire monolith version")
    monolith_hash: str = Field(
        ...,
        description="SHA-256 hash of questionnaire_monolith.json",
        pattern=r"^[a-f0-9]{64}$"
    )
    plan_name: str = Field(..., description="Development plan name", min_length=1)
    total_questions: int = Field(..., description="Total number of questions", ge=0)
    questions_analyzed: int = Field(..., description="Number of questions analyzed", ge=0)
    metadata: dict[str, Any] = Field(default_factory=dict, description="Additional metadata")
    correlation_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="UUID for request correlation"
    )

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

    @field_validator('questions_analyzed')
    @classmethod
    def validate_analyzed_count(cls, v: int, info) -> int:
        """Validate analyzed count doesn't exceed total."""
        # Note: 'total_questions' may not be available yet during construction
        # This is validated in post_init if needed
```

```
        if v < 0:
            raise ReportValidationError(
                "questions_analyzed must be non-negative",
                details={'questions_analyzed': v},
                stage="metadata_validation"
            )
        return v


class QuestionAnalysis(BaseModel):
    """Enhanced analysis result for a single micro question."""

    model_config = ConfigDict(
        frozen=True,
        extra='forbid',
        validate_assignment=True,
    )

    question_id: str = Field(..., description="Question identifier", min_length=1)
    question_global: int = Field(..., description="Global question number", ge=1, le=500)
    base_slot: str = Field(..., description="Base slot identifier")
    scoring_modality: str | None = Field(default=None, description="Scoring modality")
    score: float | None = Field(default=None, description="Question score", ge=0.0, le=1.0)
    evidence: list[str] = Field(default_factory=list, description="Evidence list")
    patterns_applied: list[str] = Field(default_factory=list, description="Applied pattern IDs")
    recommendation: str | None = Field(default=None, description="Analysis recommendation")
    metadata: dict[str, Any] = Field(default_factory=dict, description="Additional metadata")

    @field_validator('score')
    @classmethod
    def validate_score_bounds(cls, v: float | None) -> float | None:
        """Validate score is within bounds if present."""
        if v is not None:
            min_score = 0.0
            max_score = 1.0
            if not (min_score <= v <= max_score):
                raise ReportValidationError(
                    f"Score must be in [{min_score}, {max_score}], got {v}",
                    details={'score': v, 'min': min_score, 'max': max_score},
                    stage="question_validation"
                )
            # Round to avoid floating point precision issues
            return round(v, 6)
            return round(v, 6)
        return v


class Recommendation(BaseModel):
    """Structured recommendation with type and severity classification."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    type: str = Field(..., description="Recommendation type (RISK, PRIORITY, OMISSION, etc.)")
    severity: str = Field(..., description="Severity level (CRITICAL, HIGH, MEDIUM, LOW, INFO)")
    description: str = Field(..., description="Actionable recommendation text")
    source: str = Field(default="macro", description="Source of recommendation (micro, meso, macro)")
```

```
    @classmethod
    def from_string(cls, text: str, source: str = "macro") -> "Recommendation":
        """Parse recommendation string into structured object."""
        # Expected format: "TYPE_LEVEL: Description"
        # e.g., "CRITICAL_RISK: Immediate intervention required"
        if ":" in text:
            prefix, desc = text.split(":", 1)
            desc = desc.strip()

            # Parse prefix like "CRITICAL_RISK" -> severity="CRITICAL", type="RISK"
            parts = prefix.split("_")
            if len(parts) >= 2:
                severity = parts[0]
                rec_type = "_".join(parts[1:])
            else:
                severity = "INFO"
                rec_type = prefix
        else:
            severity = "INFO"
            rec_type = "GENERAL"
            desc = text

        return cls(
            type=rec_type,
            severity=severity,
            description=desc,
            source=source
        )


class MesoCluster(BaseModel):
    """Validated meso-level cluster analysis."""

    model_config = ConfigDict(frozen=True, extra='forbid')

    cluster_id: str = Field(..., min_length=1)
    raw_meso_score: float = Field(..., ge=0.0, le=1.0)
    adjusted_score: float = Field(..., ge=0.0, le=1.0)

    # Penalties
    dispersion_penalty: float = Field(..., ge=0.0, le=1.0)
    peer_penalty: float = Field(..., ge=0.0, le=1.0)
    total_penalty: float = Field(..., ge=0.0, le=1.0)

    # Metrics
    dispersion_metrics: dict[str, float] = Field(default_factory=dict)
    micro_scores: list[float] = Field(default_factory=list)

    metadata: dict[str, Any] = Field(default_factory=dict)


class MacroSummary(BaseModel):
    """Validated macro-level portfolio analysis."""

    model_config = ConfigDict(frozen=True, extra='forbid')
```

```

    overall_posterior: float = Field(..., ge=0.0, le=1.0)
    adjusted_score: float = Field(..., ge=0.0, le=1.0)

    # Penalties
    coverage_penalty: float = Field(..., ge=0.0, le=1.0)
    dispersion_penalty: float = Field(..., ge=0.0, le=1.0)
    contradiction_penalty: float = Field(..., ge=0.0, le=1.0)
    total_penalty: float = Field(..., ge=0.0, le=1.0)

    # Counts
    contradiction_count: int = Field(..., ge=0)

    # Recommendations
    recommendations: list[Recommendation] = Field(default_factory=list)

    metadata: dict[str, Any] = Field(default_factory=dict)


class AnalysisReport(BaseModel):
    """Enhanced complete policy analysis report with cryptographic verification."""

    model_config = ConfigDict(
        frozen=True,
        extra='forbid',
        validate_assignment=True,
    )

    metadata: ReportMetadata = Field(..., description="Report metadata")
    micro_analyses: list[QuestionAnalysis] = Field(..., description="Micro-level analyses")
    meso_clusters: dict[str, MesoCluster] = Field(default_factory=dict, description="Meso-level clusters")
    macro_summary: MacroSummary | None = Field(default=None, description="Macro-level summary")
    evidence_chain_hash: str | None = Field(
        default=None,
        description="Evidence chain hash",
        pattern=r"^[a-f0-9]{64}$"
    )
    report_digest: str | None = Field(
        default=None,
        description="SHA-256 digest of report content",
        pattern=r"^[a-f0-9]{64}$"
    )

    @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert report to dictionary for JSON serialization."""
        report_dict = {
            'metadata': self.metadata.model_dump(),
            'micro_analyses': [q.model_dump() for q in self.micro_analyses],
            'meso_clusters': {k: v.model_dump() for k, v in self.meso_clusters.items()},
            'macro_summary': self.macro_summary.model_dump() if self.macro_summary else None,
            'evidence_chain_hash': self.evidence_chain_hash,
            'report_digest': self.report_digest
        }
        return report_dict

```

```
    @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.compute_digest")
    def compute_digest(self) -> str:
        """Compute cryptographic digest of report content."""
        # Create deterministic representation without the digest field
        content = {
            'metadata': self.metadata.model_dump(),
            'micro_analyses': [q.model_dump() for q in self.micro_analyses],
            'meso_clusters': {k: v.model_dump() for k, v in self.meso_clusters.items()},
            'macro_summary': self.macro_summary.model_dump() if self.macro_summary else None,
            'evidence_chain_hash': self.evidence_chain_hash
        }
        return compute_content_digest(content)

    @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.verify_digest")
    def verify_digest(self) -> bool:
        """Verify report digest matches computed hash."""
        if self.report_digest is None:
            return False
        computed = self.compute_digest()
        return computed == self.report_digest


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

class ReportLogger:
    """Structured JSON logger for report assembly operations."""

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

    def log_operation(
        self,
        operation: str,
        correlation_id: str,
        success: bool,
        latency_ms: float,
        **kwargs: Any
    ) -> None:
        """Log operation event with structured data."""
        log_entry = {
            "event": "report_operation",
            "operation": operation,
            "correlation_id": correlation_id,
            "success": success,
            "latency_ms": round(latency_ms, 3),
            "timestamp_utc": utc_now_iso(),
        }
        log_entry.update(kwargs)

        self.logger.info(json.dumps(log_entry, sort_keys=True))

    def log_validation(
```

```
        self,
        item_type: str,
        correlation_id: str,
        success: bool,
        error: str | None = None,
        **kwargs: Any
    ) -> None:
        """Log validation event."""
        log_entry = {
            "event": "report_validation",
            "item_type": item_type,
            "correlation_id": correlation_id,
            "success": success,
            "timestamp_utc": utc_now_iso(),
        }
        if error:
            log_entry["error"] = error
        log_entry.update(kwargs)

        self.logger.info(json.dumps(log_entry, sort_keys=True))


# ============================================================================
# REPORT ASSEMBLER
# ============================================================================

class ReportAssembler:
    """
    Assembles comprehensive policy analysis reports.

    This class demonstrates proper architectural patterns:
    - Dependency injection for all external resources
    - No direct file I/O (delegates to factory)
    - Pattern extraction via QuestionnaireResourceProvider
    - Cryptographic traceability via SHA-256 digests
    - Domain-specific exceptions with structured payloads
    - Pydantic contract validation
    - Structured JSON logging
    """

    def __init__(
        self,
        questionnaire_provider,
        evidence_registry=None,
        qmcm_recorder=None,
        orchestrator=None
    ) -> None:
        """
        Initialize report assembler.

        Args:
            questionnaire_provider: QuestionnaireResourceProvider instance (required)
            evidence_registry: EvidenceRegistry for traceability (optional)
            qmcm_recorder: QMCMRecorder for quality monitoring (optional)
            orchestrator: Orchestrator instance for execution results (optional)

```

```
        ARCHITECTURAL NOTE: All dependencies injected, no direct I/O.
        """
        if questionnaire_provider is None:
            raise ReportValidationError(
                "questionnaire_provider is required",
                details={'provider': None},
                stage="initialization",
                recoverable=False
            )

        self.questionnaire_provider = questionnaire_provider
        self.evidence_registry = evidence_registry
        self.qmcm_recorder = qmcm_recorder
        self.orchestrator = orchestrator
        self.report_logger = ReportLogger(__name__)

        logger.info("ReportAssembler initialized with dependency injection")

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler.assemble_report")
    def assemble_report(
        self,
        plan_name: str,
        execution_results: dict[str, Any],
        report_id: str | None = None,
        enriched_packs: dict[str, Any] | None = None
    ) -> AnalysisReport:
        """
        Assemble complete analysis report.

        Args:
            plan_name: Name of the development plan
            execution_results: Results from orchestrator execution
            report_id: Optional report identifier

        Returns:
            Structured AnalysisReport with full traceability

        Raises:
            ReportValidationError: If input validation fails
            ReportIntegrityError: If hash computation fails
        """
        import time
        start_time = time.time()

        # Input validation
        if not plan_name or not isinstance(plan_name, str):
            raise ReportValidationError(
                "plan_name must be a non-empty string",
                details={'plan_name': plan_name, 'type': type(plan_name).__name__},
                stage="input_validation"
            )

        if not isinstance(execution_results, dict):
            raise ReportValidationError(
                "execution_results must be a dictionary",
                details={'type': type(execution_results).__name__},
```

```
                stage="input_validation"
            )

        # Generate report ID if not provided
        if report_id is None:
            timestamp = datetime.now(timezone.utc).strftime("%Y%m%d_%H%M%S")
            report_id = f"report_{plan_name}_{timestamp}"

        correlation_id = str(uuid.uuid4())

        try:
            # Get questionnaire data and compute hash
            questionnaire_data = self.questionnaire_provider.get_data()

            if not isinstance(questionnaire_data, dict):
                raise ReportIntegrityError(
                    "Invalid questionnaire data format",
                    details={'type': type(questionnaire_data).__name__},
                    stage="questionnaire_loading"
                )

            # Import hash utility for content verification
            from farfan_pipeline.utils.hash_utils import compute_hash
            monolith_hash = compute_hash(questionnaire_data)

            # Validate hash format
            if not isinstance(monolith_hash, str) or len(monolith_hash) != 64:
                raise ReportIntegrityError(
                    "Invalid monolith hash format",
                    details={'hash': monolith_hash, 'length': len(monolith_hash) if isinstance(monolith_hash, str) else 0},
                    stage="hash_computation"
                )

            # Extract metadata with defensive checks
            version = questionnaire_data.get('version', 'unknown')
            blocks = questionnaire_data.get('blocks', {})
            if not isinstance(blocks, dict):
                raise ReportValidationError(
                    "questionnaire blocks must be a dictionary",
                    details={'type': type(blocks).__name__},
                    stage="data_extraction"
                )

            micro_questions = blocks.get('micro_questions', [])
            if not isinstance(micro_questions, list):
                raise ReportValidationError(
                    "micro_questions must be a list",
                    details={'type': type(micro_questions).__name__},
                    stage="data_extraction"
                )

            # Create report metadata with Pydantic validation
            metadata = ReportMetadata(
                report_id=report_id,
                generated_at=utc_now_iso(),
                monolith_version=version,
```

```
            monolith_hash=monolith_hash,
            plan_name=plan_name,
            total_questions=len(micro_questions),
            questions_analyzed=len(execution_results.get('questions', {})),
            correlation_id=correlation_id
        )

        # Assemble micro analyses
        micro_analyses = self._assemble_micro_analyses(
            micro_questions,
            execution_results,
            correlation_id
        )

        # Assemble meso clusters
        meso_clusters = self._assemble_meso_clusters(execution_results)

        # Assemble macro summary
        macro_summary = self._assemble_macro_summary(execution_results)

        # Get evidence chain hash if available
        evidence_chain_hash = None
        if self.evidence_registry is not None:
            records = self.evidence_registry.records
            if records:
                evidence_chain_hash = records[-1].entry_hash

        # JOBFRONT 9: Compute signal usage summary if enriched_packs provided
        if enriched_packs:
            signal_usage = self._compute_signal_usage_summary(
                execution_results,
                enriched_packs
            )
            # Add to metadata
            if metadata.metadata is None:
                # metadata.metadata is immutable, need to recreate
                from dataclasses import replace
                new_metadata_dict = {
                    'signal_version': '1.0.0',
                    'total_patterns_available': signal_usage['total_patterns_available'],
                    'total_patterns_used': signal_usage['total_patterns_used'],
                    'signal_usage_summary': signal_usage
                }
                metadata = ReportMetadata(
                    report_id=metadata.report_id,
                    generated_at=metadata.generated_at,
                    monolith_version=metadata.monolith_version,
                    monolith_hash=metadata.monolith_hash,
                    plan_name=metadata.plan_name,
                    total_questions=metadata.total_questions,
                    questions_analyzed=metadata.questions_analyzed,
                    metadata=new_metadata_dict,
                    correlation_id=metadata.correlation_id
                )

        # Create report and compute digest
```

```
            report = AnalysisReport(
                metadata=metadata,
                micro_analyses=micro_analyses,
                meso_clusters=meso_clusters,
                macro_summary=macro_summary,
                evidence_chain_hash=evidence_chain_hash,
                report_digest=None  # Will be computed
            )

            # Compute and attach digest
            report_digest = report.compute_digest()
            report = AnalysisReport(
                metadata=metadata,
                micro_analyses=micro_analyses,
                meso_clusters=meso_clusters,
                macro_summary=macro_summary,
                evidence_chain_hash=evidence_chain_hash,
                report_digest=report_digest
            )

            latency_ms = (time.time() - start_time) * 1000

            # Structured logging
            self.report_logger.log_operation(
                operation="assemble_report",
                correlation_id=correlation_id,
                success=True,
                latency_ms=latency_ms,
                report_id=report_id,
                question_count=len(micro_analyses),
                monolith_hash=monolith_hash[:16],
                report_digest=report_digest[:16]
            )

            logger.info(
                f"Report assembled: {report_id} "
                f"({len(micro_analyses)} questions, hash: {monolith_hash[:16]}...)"
            )

            return report

        except ReportAssemblyException:
            # Re-raise our domain exceptions
            raise
        except Exception as e:
            # Wrap unexpected exceptions
            raise ReportAssemblyException(
                f"Unexpected error during report assembly: {str(e)}",
                details={'error_type': type(e).__name__, 'error': str(e)},
                stage="assembly",
                recoverable=False
            ) from e

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_micro_analyses")
    def _assemble_micro_analyses(
        self,
```

```
        micro_questions: list[dict[str, Any]],
        execution_results: dict[str, Any],
        correlation_id: str
    ) -> list[QuestionAnalysis]:
        """Assemble micro-level question analyses with validation."""
        analyses = []
        question_results = execution_results.get('questions', {})

        if not isinstance(question_results, dict):
            raise ReportValidationError(
                "execution_results.questions must be a dictionary",
                details={'type': type(question_results).__name__},
                stage="micro_analysis"
            )

        for question in micro_questions:
            if not isinstance(question, dict):
                logger.warning(f"Skipping invalid question entry: {type(question).__name__}")
                continue

            question_id = question.get('question_id', '')
            if not question_id:
                logger.warning("Skipping question with missing question_id")
                continue

            result = question_results.get(question_id, {})

            # Extract patterns applied using QuestionnaireResourceProvider
            patterns = self.questionnaire_provider.get_patterns_by_question(question_id)
            pattern_names = [p.get('pattern_id', '') for p in patterns] if patterns else []

            try:
                # Pydantic validation
                analysis = QuestionAnalysis(
                    question_id=question_id,
                    question_global=question.get('question_global', 0),
                    base_slot=question.get('base_slot', ''),
                    scoring_modality=question.get('scoring', {}).get('modality') if isinstance(question.get('scoring'), dict) else None,
                    score=result.get('score'),
                    evidence=result.get('evidence', []) if isinstance(result.get('evidence'), list) else [],
                    patterns_applied=pattern_names,
                    recommendation=result.get('recommendation'),
                    metadata={
                        'dimension': question.get('dimension'),
                        'policy_area': question.get('policy_area')
                    }
                )
                analyses.append(analysis)

                self.report_logger.log_validation(
                    item_type="question_analysis",
                    correlation_id=correlation_id,
                    success=True,
                    question_id=question_id
                )

```

```
            except Exception as e:
                # Log validation failure but continue
                self.report_logger.log_validation(
                    item_type="question_analysis",
                    correlation_id=correlation_id,
                    success=False,
                    error=str(e),
                    question_id=question_id
                )
                logger.error(f"Failed to create QuestionAnalysis for {question_id}: {e}")

        return analyses

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_meso_clusters")
    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_meso_clusters")
    def _assemble_meso_clusters(
        self,
        execution_results: dict[str, Any]
    ) -> dict[str, MesoCluster]:
        """Assemble meso-level cluster analyses with strict validation."""
        raw_clusters = execution_results.get('meso_clusters', {})

        # Handle list format from Bayesian orchestrator
        if isinstance(raw_clusters, list):
            # Convert list of objects to dict keyed by cluster_id
            cluster_dict = {}
            for item in raw_clusters:
                # Handle both dicts and objects (if coming from dataclasses)
                if hasattr(item, '__dict__'):
                    data = item.__dict__
                elif isinstance(item, dict):
                    data = item
                else:
                    continue

                c_id = data.get('cluster_id')
                if c_id:
                    cluster_dict[c_id] = data
            raw_clusters = cluster_dict

        if not isinstance(raw_clusters, dict):
            logger.warning(f"meso_clusters is not a dict/list, got {type(raw_clusters).__name__}")
            return {}

        validated_clusters = {}
        for cluster_id, data in raw_clusters.items():
            try:
                # Ensure data is a dict
                if hasattr(data, '__dict__'):
                    data = data.__dict__

                if not isinstance(data, dict):
                    continue

                cluster = MesoCluster(
                    cluster_id=str(data.get('cluster_id', cluster_id)),
```

```
                    raw_meso_score=float(data.get('raw_meso_score', 0.0)),
                    adjusted_score=float(data.get('adjusted_score', 0.0)),
                    dispersion_penalty=float(data.get('dispersion_penalty', 0.0)),
                    peer_penalty=float(data.get('peer_penalty', 0.0)),
                    total_penalty=float(data.get('total_penalty', 0.0)),
                    dispersion_metrics=data.get('dispersion_metrics', {}),
                    micro_scores=data.get('micro_scores', []),
                    metadata=data.get('metadata', {})
                )
                validated_clusters[cluster_id] = cluster
            except Exception as e:
                logger.error(f"Failed to validate meso cluster {cluster_id}: {e}")

        return validated_clusters

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_macro_summary")
    def _assemble_macro_summary(
        self,
        execution_results: dict[str, Any]
    ) -> MacroSummary | None:
        """Assemble macro-level summary with strict validation and recommendation wiring."""
        raw_macro = execution_results.get('macro_summary', {})

        # Handle object format
        if hasattr(raw_macro, '__dict__'):
            raw_macro = raw_macro.__dict__

        if not isinstance(raw_macro, dict) or not raw_macro:
            logger.warning("macro_summary is missing or invalid")
            return None

        try:
            # Parse recommendations
            raw_recs = raw_macro.get('recommendations', [])
            validated_recs = []
            for rec in raw_recs:
                if isinstance(rec, str):
                    validated_recs.append(Recommendation.from_string(rec))
                elif isinstance(rec, dict):
                    # Already structured?
                    try:
                        validated_recs.append(Recommendation(**rec))
                    except:
                        pass

            return MacroSummary(
                overall_posterior=float(raw_macro.get('overall_posterior', 0.0)),
                adjusted_score=float(raw_macro.get('adjusted_score', 0.0)),
                coverage_penalty=float(raw_macro.get('coverage_penalty', 0.0)),
                dispersion_penalty=float(raw_macro.get('dispersion_penalty', 0.0)),
                contradiction_penalty=float(raw_macro.get('contradiction_penalty', 0.0)),
                total_penalty=float(raw_macro.get('total_penalty', 0.0)),
                contradiction_count=int(raw_macro.get('contradiction_count', 0)),
                recommendations=validated_recs,
                metadata=raw_macro.get('metadata', {})
            )
```

```
        except Exception as e:
            logger.error(f"Failed to validate macro summary: {e}")
            return None

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler.export_report")
    def export_report(
        self,
        report: AnalysisReport,
        output_path: Path,
        format: str = 'json'
    ) -> None:
        """
        Export report to file.

        Args:
            report: AnalysisReport to export
            output_path: Path to output file
            format: Output format ('json' or 'markdown')

        Raises:
            ReportExportError: If export fails
            ReportValidationError: If format is unsupported

        NOTE: This delegates I/O to factory for architectural compliance.
        """
        import time
        start_time = time.time()
        correlation_id = report.metadata.correlation_id

        try:
            # Delegate to factory for I/O
            from farfan_pipeline.analysis.factory import save_json, write_text_file

            if format == 'json':
                save_json(report.to_dict(), str(output_path))
            elif format == 'markdown':
                markdown = self._format_as_markdown(report)
                write_text_file(markdown, str(output_path))
            else:
                raise ReportValidationError(
                    f"Unsupported format: {format}",
                    details={'format': format, 'supported': ['json', 'markdown']},
                    stage="export"
                )

            latency_ms = (time.time() - start_time) * 1000

            self.report_logger.log_operation(
                operation="export_report",
                correlation_id=correlation_id,
                success=True,
                latency_ms=latency_ms,
                output_path=str(output_path),
                format=format
            )

```

```
            logger.info(f"Report exported to {output_path} in {format} format")

        except ReportValidationError:
            raise
        except Exception as e:
            raise ReportExportError(
                f"Failed to export report: {str(e)}",
                details={'output_path': str(output_path), 'format': format, 'error': str(e)},
                stage="export",
                recoverable=True
            ) from e

    @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown")
    def _format_as_markdown(self, report: AnalysisReport) -> str:
        """Format report as Markdown with externalized parameters."""
        # Externalized parameters
        # Load from calibration system if available
        if _PARAM_LOADER:
            preview_count = _PARAM_LOADER.get("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown").get("preview_question_count", 10)
            hash_preview_length = _PARAM_LOADER.get("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown").get("hash_preview_length", 16)

        else:
            preview_count = 10
            hash_preview_length = 16

        lines = [
            f"# Policy Analysis Report: {report.metadata.plan_name}\n",
            f"**Report ID:** {report.metadata.report_id}\n",
            f"**Generated:** {report.metadata.generated_at}\n",
            f"**Monolith Version:** {report.metadata.monolith_version}\n",
            f"**Monolith Hash:** {report.metadata.monolith_hash[:hash_preview_length]}...\n",
            f"**Questions Analyzed:** {report.metadata.questions_analyzed}/{report.metadata.total_questions}\n",
        ]

        if report.report_digest:
            lines.append(f"**Report Digest:** {report.report_digest[:hash_preview_length]}...\n")

        lines.append("\n## Micro-Level Analyses\n")

        for analysis in report.micro_analyses[:preview_count]:
            lines.append(f"\n### {analysis.question_id}\n")
            lines.append(f"- **Slot:** {analysis.base_slot}\n")
            lines.append(f"- **Score:** {analysis.score}\n")
            lines.append(f"- **Patterns:** {', '.join(analysis.patterns_applied)}\n")

        if len(report.micro_analyses) > preview_count:
            lines.append(f"\n_...and {len(report.micro_analyses) - preview_count} more questions_\n")

        lines.append("\n## Meso-Level Clusters\n")
        for cid, cluster in report.meso_clusters.items():
            lines.append(f"\n### Cluster {cid}\n")
            lines.append(f"- **Score:** {cluster.adjusted_score:.4f} (Raw: {cluster.raw_meso_score:.4f})\n")
            lines.append(f"- **Penalties:** Total {cluster.total_penalty:.4f} (Dispersion: {cluster.dispersion_penalty:.4f}, Peer: {cluster.peer_penalty:.4f})\n")

        if report.macro_summary:
```

```
            lines.append("\n## Macro Summary\n")
            lines.append(f"- **Overall Score:** {report.macro_summary.adjusted_score:.4f}\n")
            lines.append(f"- **Contradictions:** {report.macro_summary.contradiction_count}\n")

            lines.append("\n### Recommendations\n")
            for rec in report.macro_summary.recommendations:
                icon = "ð\237\224´" if "CRITICAL" in rec.severity else "ð\237\237 " if "HIGH" in rec.severity else "ð\237\237¡"
                lines.append(f"- {icon} **{rec.type}** ({rec.severity}): {rec.description}\n")
        else:
            lines.append("\n## Macro Summary\n")
            lines.append("_No macro summary available_\n")

        if report.evidence_chain_hash:
            lines.append(f"\n**Evidence Chain Hash:** {report.evidence_chain_hash[:hash_preview_length]}...\n")

        return "".join(lines)

    def _compute_signal_usage_summary(
        self,
        execution_results: dict[str, Any],
        enriched_packs: dict[str, Any]
    ) -> dict[str, Any]:
        """
        Compute signal usage summary for report provenance (JOBFRONT 9).

        Args:
            execution_results: Results from orchestrator execution
            enriched_packs: Dictionary of EnrichedSignalPack by policy_area_id

        Returns:
            Signal usage summary with patterns, completeness, validation failures
        """
        micro_results = execution_results.get("micro_results", {})

        total_patterns_available = sum(len(pack.patterns) for pack in enriched_packs.values())
        total_patterns_used = 0
        by_policy_area = {}
        completeness_scores = []
        validation_failures = []

        for question_id, result in micro_results.items():
            policy_area = result.get("policy_area_id")
            if not policy_area or policy_area not in enriched_packs:
                continue

            patterns_used = result.get("patterns_used", [])
            completeness = result.get("completeness", 1.0)
            validation = result.get("validation", {})

            total_patterns_used += len(patterns_used)
            completeness_scores.append(completeness)

            # Track validation failures
            if validation.get("status") == "failed" or validation.get("contract_failed"):
                validation_failures.append({
                    "question_id": question_id,
```

```
                    "policy_area": policy_area,
                    "error_code": validation.get("errors", [{}])[0].get("error_code") if validation.get("errors") else None,
                    "remediation": validation.get("errors", [{}])[0].get("remediation") if validation.get("errors") else None
                })

            # Aggregate by policy area
            if policy_area not in by_policy_area:
                by_policy_area[policy_area] = {
                    "patterns_available": len(enriched_packs[policy_area].patterns),
                    "patterns_used": 0,
                    "questions_analyzed": 0,
                    "avg_completeness": 0.0
                }

            by_policy_area[policy_area]["patterns_used"] += len(patterns_used)
            by_policy_area[policy_area]["questions_analyzed"] += 1

        # Compute averages
        for pa_id, summary in by_policy_area.items():
            pa_results = [r for r in micro_results.values() if r.get("policy_area_id") == pa_id]
            completeness_values = [r.get("completeness", 1.0) for r in pa_results]
            summary["avg_completeness"] = sum(completeness_values) / len(completeness_values) if completeness_values else 0.0

        return {
            "total_patterns_available": total_patterns_available,
            "total_patterns_used": total_patterns_used,
            "by_policy_area": by_policy_area,
            "avg_completeness": sum(completeness_scores) / len(completeness_scores) if completeness_scores else 0.0,
            "validation_failures": validation_failures
        }


# ============================================================================
# FACTORY FUNCTIONS
# ============================================================================

def create_report_assembler(
    questionnaire_provider,
    evidence_registry=None,
    qmcm_recorder=None,
    orchestrator=None
) -> ReportAssembler:
    """
    Factory function to create ReportAssembler with dependencies.

    Args:
        questionnaire_provider: QuestionnaireResourceProvider instance
        evidence_registry: Optional EvidenceRegistry
        qmcm_recorder: Optional QMCMRecorder
        orchestrator: Optional Orchestrator

    Returns:
        Configured ReportAssembler

    Raises:
        ReportValidationError: If required dependencies are missing
```

```
    """
    return ReportAssembler(
        questionnaire_provider=questionnaire_provider,
        evidence_registry=evidence_registry,
        qmcm_recorder=qmcm_recorder,
        orchestrator=orchestrator
    )


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

__all__ = [
    # Exceptions
    ’ReportAssemblyException’,
    ’ReportValidationError’,
    ’ReportIntegrityError’,
    ’ReportExportError’,
    # Contracts
    ’ReportMetadata’,
    ’QuestionAnalysis’,
    ’AnalysisReport’,
    # Main Classes
    ’ReportAssembler’,
    ’ReportLogger’,
    # Factory Functions
    ’create_report_assembler’,
    # Utilities
    ’compute_content_digest’,
    ’utc_now_iso’,
]


# ============================================================================
# IN-SCRIPT VALIDATION
# ============================================================================

if __name__ == "__main__":
    print("=" * 70)
    print("Report Assembly Module - Validation Suite")
    print("=" * 70)

    # Test 1: Domain-specific exceptions
    print("\n1. Testing domain-specific exceptions:")
    try:
        raise ReportValidationError(
            "Test validation error",
            details={’field’: ’test’},
            stage="validation"
        )
    except ReportValidationError as e:
        print(f"   â\234\223 ReportValidationError: {e.event_id[:8]}... - {e.message}")
        print(f"   â\234\223 Structured dict: {list(e.to_dict().keys())}")

    # Test 2: Pydantic contract validation
```

```
    print("\n2. Testing Pydantic contract validation:")
    try:
        # Invalid hash (not 64 chars)
        ReportMetadata(
            report_id="test-001",
            monolith_version="1.0",
            monolith_hash="invalid",
            plan_name="Test Plan",
            total_questions=10,
            questions_analyzed=5
        )
        print("   â\234\227 Expected validation error for invalid hash")
    except Exception as e:
        print(f"   â\234\223 Caught validation error: {type(e).__name__}")

    # Valid metadata
    valid_hash = "a" * 64
    metadata = ReportMetadata(
        report_id="test-001",
        monolith_version="1.0",
        monolith_hash=valid_hash,
        plan_name="Test Plan",
        total_questions=10,
        questions_analyzed=5
    )
    print(f"   â\234\223 Valid metadata created: {metadata.report_id}")

    # Test 3: Cryptographic digest
    print("\n3. Testing cryptographic digest:")
    test_content = {"key": "value", "number": 42}
    digest = compute_content_digest(test_content)
    print(f"   â\234\223 Digest computed: {digest[:16]}... (length: {len(digest)})")
    assert len(digest) == 64, "Digest must be 64 characters"
    print("   â\234\223 Digest length validated")

    # Test 4: Report digest verification
    print("\n4. Testing report digest verification:")
    micro_analysis = QuestionAnalysis(
        question_id="Q001",
        question_global=1,
        base_slot="slot1",
        score=0.85
    )

    meso_cluster = MesoCluster(
        cluster_id="CL01",
        raw_meso_score=0.8,
        adjusted_score=0.75,
        dispersion_penalty=0.05,
        peer_penalty=0.0,
        total_penalty=0.05
    )

    macro_summary = MacroSummary(
        overall_posterior=0.75,
        adjusted_score=0.7,
```

```
        coverage_penalty=0.05,
        dispersion_penalty=0.0,
        contradiction_penalty=0.0,
        total_penalty=0.05,
        contradiction_count=0,
        recommendations=[
            Recommendation(type="RISK", severity="LOW", description="Monitor closely")
        ]
    )

    report = AnalysisReport(
        metadata=metadata,
        micro_analyses=[micro_analysis],
        meso_clusters={"CL01": meso_cluster},
        macro_summary=macro_summary
    )

    report_digest = report.compute_digest()
    print(f"   â\234\223 Report digest: {report_digest[:16]}...")

    # Create report with digest
    report_with_digest = AnalysisReport(
        metadata=metadata,
        micro_analyses=[micro_analysis],
        meso_clusters={"CL01": meso_cluster},
        macro_summary=macro_summary,
        report_digest=report_digest
    )

    is_valid = report_with_digest.verify_digest()
    print(f"   â\234\223 Digest verification: {is_valid}")

    # Test 5: Structured logging
    print("\n5. Testing structured logging:")
    test_logger = ReportLogger("test")
    test_logger.log_operation(
        operation="test_operation",
        correlation_id=metadata.correlation_id,
        success=True,
        latency_ms=12.345,
        custom_field="test_value"
    )
    print("   â\234\223 Structured log emitted")

    print("\n" + "=" * 70)
    print("All validation tests passed!")
    print("=" * 70)



================================================================================
FILE: src/farfan_pipeline/analysis/retry_handler.py
================================================================================

from enum import Enum
import time
```

```
import logging
from functools import wraps

class DependencyType(Enum):
    SPACY_MODEL = "spaCy_model"
    PDF_PARSER = "pdf_parser"

def get_retry_handler():
    return RetryHandler()

class RetryHandler:
    def __init__(self, max_retries=3, delay=1):
        self.max_retries = max_retries
        self.delay = delay
        self.logger = logging.getLogger(self.__class__.__name__)

    def with_retry(self, dependency_type, operation_name, exceptions):
        def decorator(func):
            @wraps(func)
            def wrapper(*args, **kwargs):
                retries = 0
                while retries < self.max_retries:
                    try:
                        return func(*args, **kwargs)
                    except exceptions as e:
                        retries += 1
                        self.logger.warning(
                            f"Operation '{operation_name}' for dependency '{dependency_type.value}' failed. "
                            f"Retrying ({retries}/{self.max_retries})... Error: {e}"
                        )
                        time.sleep(self.delay)
                self.logger.error(
                    f"Operation '{operation_name}' for dependency '{dependency_type.value}' "
                    f"failed after {self.max_retries} retries."
                )
                raise
            return wrapper
        return decorator



================================================================================
FILE: src/farfan_pipeline/analysis/scoring/__init__.py
================================================================================

"""
Scoring Package

Implements TYPE_A through TYPE_F scoring modalities with strict validation
and reproducible results.

NOTE: Evidence and MicroQuestionScorer are NOT in this package.
They exist in the parent MODULE: farfan_core/analysis/scoring.py
Import them directly from there: `from farfan_core.analysis.scoring import Evidence`
"""

```

```
# Import from this package's scoring.py
from farfan_pipeline.analysis.scoring.scoring import (
    EvidenceStructureError,
    ModalityConfig,
    ModalityValidationError,
    QualityLevel,
    ScoredResult,
    ScoringError,
    ScoringModality,
    ScoringValidator,
    apply_scoring,
    determine_quality_level,
)

__all__ = [
    "EvidenceStructureError",
    "ModalityConfig",
    "ModalityValidationError",
    "QualityLevel",
    "ScoredResult",
    "ScoringError",
    "ScoringModality",
    "ScoringValidator",
    "apply_scoring",
    "determine_quality_level",
]

# ARCHITECTURAL NOTE FOR MAINTAINERS:
# Evidence and MicroQuestionScorer live in farfan_core/analysis/scoring.py (module)
# This __init__.py is for farfan_core/analysis/scoring/ (package)
# These are SEPARATE namespaces. Import Evidence from the module directly.



================================================================================
FILE: src/farfan_pipeline/analysis/scoring/scoring.py
================================================================================

"""
Scoring Module - TYPE_A through TYPE_F Modality Implementation

This module implements the scoring system for the SAAAAAA policy analysis framework.
It provides:
- Application of 6 scoring modalities (TYPE_A through TYPE_F)
- Validation of evidence structure vs modality
- Assignment of quality levels
- Structured logging with strict abortability
- Reproducible ScoredResult outputs

Preconditions:
- Evidence and modality must be declared
- Evidence structure must match modality requirements

Invariants:
- Score range is maintained per modality definition
- Evidence structure is validated before scoring
```

```

Postconditions:
- ScoredResult is reproducible with same inputs
- No fallback or partial heuristic scoring
"""

from __future__ import annotations

import hashlib
import json
import logging
import math
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from decimal import ROUND_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, Decimal, InvalidOperation
from enum import Enum
from numbers import Real
from typing import Any, ClassVar
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

class ScoringModality(Enum):
    """Scoring modality types."""
    TYPE_A = "TYPE_A"  # Bayesian: Numerical claims, gaps, risks
    TYPE_B = "TYPE_B"  # DAG: Causal chains, ToC completeness
    TYPE_C = "TYPE_C"  # Coherence: Inverted contradictions
    TYPE_D = "TYPE_D"  # Pattern: Baseline data, formalization
    TYPE_E = "TYPE_E"  # Financial: Budget traceability
    TYPE_F = "TYPE_F"  # Beach: Mechanism inference, plausibility

class QualityLevel(Enum):
    """Quality level classifications."""
    EXCELENTE = "EXCELENTE"
    BUENO = "BUENO"
    ACEPTABLE = "ACEPTABLE"
    INSUFICIENTE = "INSUFICIENTE"

class ScoringError(Exception):
    """Base exception for scoring errors."""
    pass

class ModalityValidationError(ScoringError):
    """Exception raised when evidence structure doesn’t match modality requirements."""
    pass

class EvidenceStructureError(ScoringError):
    """Exception raised when evidence structure is invalid."""
    pass

@dataclass(frozen=True)
class ScoredResult:
    """
    Reproducible scored result for a question.

```

```
    Attributes:
        question_global: Global question number (1-300)
        base_slot: Question slot identifier
        policy_area: Policy area ID (PA01-PA10)
        dimension: Dimension ID (DIM01-DIM06)
        modality: Scoring modality used (TYPE_A through TYPE_F)
        score: Raw score value
        normalized_score: Normalized score (0-1)
        quality_level: Quality level classification
        evidence_hash: SHA-256 hash of evidence for reproducibility
        metadata: Additional scoring metadata
        timestamp: ISO timestamp of scoring
    """
    question_global: int
    base_slot: str
    policy_area: str
    dimension: str
    modality: str
    score: float
    normalized_score: float
    quality_level: str
    evidence_hash: str
    metadata: dict[str, Any] = field(default_factory=dict)
    timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat().replace("+00:00", "Z"))

    @calibrated_method("farfan_core.analysis.scoring.scoring.ScoredResult.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary representation."""
        return asdict(self)

    @staticmethod
    def compute_evidence_hash(evidence: dict[str, Any]) -> str:
        """
        Compute reproducible hash of evidence.

        Args:
            evidence: Evidence dictionary

        Returns:
            SHA-256 hash as hex string
        """
        canonical = json.dumps(evidence, ensure_ascii=False, sort_keys=True, separators=(",", ":"))
        return hashlib.sha256(canonical.encode("utf-8")).hexdigest()

@dataclass
class ModalityConfig:
    """
    Configuration for a scoring modality.

    Attributes:
        name: Modality name
        description: Modality description
        score_range: Min and max score values
        rounding_mode: Rounding mode (half_up, bankers, truncate)
        rounding_precision: Decimal precision for rounding
        required_evidence_keys: Required keys in evidence
```

```
        expected_elements: Expected number of elements (if applicable)
        deterministic: Whether scoring is deterministic
    """
    name: str
    description: str
    score_range: tuple[float, float]
    rounding_mode: str = "half_up"
    rounding_precision: int = 2
    required_evidence_keys: list[str] = field(default_factory=list)
    expected_elements: int | None = None
    deterministic: bool = True

    @calibrated_method("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence")
    def validate_evidence(self, evidence: dict[str, Any]) -> None:
        """
        Validate evidence structure against modality requirements.

        Args:
            evidence: Evidence dictionary to validate

        Raises:
            EvidenceStructureError: If evidence is missing required keys
            ModalityValidationError: If evidence structure doesn't match modality
        """
        if not isinstance(evidence, dict):
            raise EvidenceStructureError(
                f"Evidence must be a dictionary, got {type(evidence).__name__}"
            )

        # Check required keys
        missing_keys = [key for key in self.required_evidence_keys if key not in evidence]
        if missing_keys:
            raise EvidenceStructureError(
                f"Evidence missing required keys for {self.name}: {missing_keys}"
            )

        # Validate expected elements if applicable
        if self.expected_elements is not None:
            elements = evidence.get("elements", [])
            if not isinstance(elements, list):
                raise ModalityValidationError(
                    f"{self.name} requires 'elements' to be a list, got {type(elements).__name__}"
                )

class ScoringValidator:
    """Validates evidence structure against modality requirements."""

    # Modality configurations
    MODALITY_CONFIGS: ClassVar[dict[ScoringModality, ModalityConfig]] = {
        ScoringModality.TYPE_A: ModalityConfig(
            name="TYPE_A",
            description="Bayesian: Numerical claims, gaps, risks",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L182_25", 0.0), 3.0),
            required_evidence_keys=["elements", "confidence"],
            expected_elements=4,
        ),
```

```
        ScoringModality.TYPE_B: ModalityConfig(
            name="TYPE_B",
            description="DAG: Causal chains, ToC completeness",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L189_25", 0.0), 3.0),
            required_evidence_keys=["elements", "completeness"],
            expected_elements=3,
        ),
        ScoringModality.TYPE_C: ModalityConfig(
            name="TYPE_C",
            description="Coherence: Inverted contradictions",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L196_25", 0.0), 3.0),
            required_evidence_keys=["elements", "coherence_score"],
            expected_elements=2,
        ),
        ScoringModality.TYPE_D: ModalityConfig(
            name="TYPE_D",
            description="Pattern: Baseline data, formalization",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L203_25", 0.0), 3.0),
            required_evidence_keys=["elements", "pattern_matches"],
            expected_elements=3,
        ),
        ScoringModality.TYPE_E: ModalityConfig(
            name="TYPE_E",
            description="Financial: Budget traceability",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L210_25", 0.0), 3.0),
            required_evidence_keys=["elements", "traceability"],
        ),
        ScoringModality.TYPE_F: ModalityConfig(
            name="TYPE_F",
            description="Beach: Mechanism inference, plausibility",
            score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L216_25", 0.0), 3.0),
            required_evidence_keys=["elements", "plausibility"],
        ),
    }

    @classmethod
    def validate(
        cls,
        evidence: dict[str, Any],
        modality: ScoringModality,
    ) -> None:
        """
        Validate evidence structure against modality.

        Args:
            evidence: Evidence dictionary
            modality: Scoring modality

        Raises:
            ModalityValidationError: If validation fails

        Note:
            This function has strict abortability - any validation failure
            will raise an exception and halt processing.
        """
        config = cls.MODALITY_CONFIGS.get(modality)
```

```
        if not config:
            raise ModalityValidationError(f"Unknown modality: {modality}")

        logger.info(f"Validating evidence for {modality.value}")

        try:
            config.validate_evidence(evidence)
            logger.info(f"â\234\223 Evidence validation passed for {modality.value}")
        except (EvidenceStructureError, ModalityValidationError) as e:
            logger.exception(f"â\234\227 Evidence validation failed for {modality.value}: {e}")
            raise

    @classmethod
    def get_config(cls, modality: ScoringModality) -> ModalityConfig:
        """Get configuration for a modality."""
        config = cls.MODALITY_CONFIGS.get(modality)
        if not config:
            raise ModalityValidationError(f"Unknown modality: {modality}")
        return config

def clamp(value: float, lower: float, upper: float) -> float:
    """Clamp *value* to the inclusive range ``[lower, upper]``."""

    if lower > upper:
        raise ValueError("Lower bound cannot exceed upper bound")

    return min(max(value, lower), upper)

def apply_rounding(
    value: float,
    mode: str = "half_up",
    precision: int = 2,
) -> float:
    """
    Apply rounding to a numeric value.

    Args:
        value: Value to round
        mode: Rounding mode (half_up, bankers, truncate)
        precision: Decimal precision

    Returns:
        Rounded value
    """
    if precision < 0:
        raise ValueError("Precision must be non-negative")

    decimal_value = Decimal(str(value))
    quantize_exp = Decimal(10) ** -precision

    if mode == "half_up":
        rounding_mode = ROUND_HALF_UP
    elif mode == "bankers":
        rounding_mode = ROUND_HALF_EVEN
    elif mode == "truncate":
        rounding_mode = ROUND_DOWN
```

```
    else:
        raise ValueError(f"Unknown rounding mode: {mode}")

    try:
        rounded = decimal_value.quantize(quantize_exp, rounding=rounding_mode)
    except InvalidOperation as exc:
        raise ValueError(f"Failed to round value {value}: {exc}") from exc

    return float(rounded)

def _validate_quality_thresholds(thresholds: dict[str, float]) -> dict[str, float]:
    """Validate custom quality thresholds.

    Returns a copy of *thresholds* with float values if validation succeeds.
    """

    if not isinstance(thresholds, dict):
        raise ValueError("Quality thresholds must be provided as a dictionary")

    required_keys = ("EXCELENTE", "BUENO", "ACEPTABLE")
    missing = [key for key in required_keys if key not in thresholds]
    if missing:
        raise ValueError(f"Missing quality thresholds for: {', '.join(missing)}")

    validated: dict[str, float] = {}
    for key in required_keys:
        value = thresholds[key]

        if isinstance(value, bool) or not isinstance(value, (int, float, Decimal, Real)):
            raise ValueError(
                f"Threshold for {key} must be a real number between 0 and 1"
            )

        numeric_value = float(value)
        if math.isnan(numeric_value) or math.isinf(numeric_value):
            raise ValueError(f"Threshold for {key} cannot be NaN or infinite")

        if not ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L335_15", 0.0) <= numeric_value <= ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L335_39", 1.0):
            raise ValueError(
                f"Threshold for {key} must be between 0 and 1 inclusive"
            )

        validated[key] = numeric_value

    if not (
        validated["EXCELENTE"] >= validated["BUENO"] >= validated["ACEPTABLE"]
    ):
        raise ValueError(
            "Quality thresholds must satisfy EXCELENTE >= BUENO >= ACEPTABLE"
        )

    return validated

def score_type_a(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
```

```
    Score TYPE_A evidence: Bayesian numerical claims, gaps, risks.

    Expects:
    - elements: List of up to 4 elements
    - confidence: Bayesian confidence score (0-1)

    Scoring:
    - Count elements (max 4)
    - Weight by confidence
    - Scale to 0-3 range

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    confidence = evidence.get("confidence", ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L372_44", 0.0))

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_A: 'elements' must be a list")

    if not isinstance(confidence, (int, float)) or not (0 <= confidence <= 1):
        raise ModalityValidationError("TYPE_A: 'confidence' must be a number between 0 and 1")

    # Count valid elements (up to expected)
    element_count = min(len(elements), config.expected_elements or 4)

    max_elements = config.expected_elements or 4
    max_score = config.score_range[1] if config.score_range else 3.0
    min_score = config.score_range[0] if config.score_range else ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L385_65", 0.0)

    # Calculate raw score: count weighted by confidence, scale to range
    max_elements = config.expected_elements if config.expected_elements is not None else 4
    scale = config.score_range[1] if config.score_range else 3.0
    raw_score = (element_count / max(1, max_elements)) * scale * confidence

    # Clamp to valid range
    score = max(min_score, min(max_score, raw_score))

    metadata = {
        "element_count": element_count,
        "confidence": confidence,
        "raw_score": raw_score,
        "expected_elements": config.expected_elements,
        "max_score": max_score,
    }

    logger.info(
        f"TYPE_A score: {score:.2f} "
        f"(elements={element_count}, confidence={confidence:.2f})"
    )
```

```

    return score, metadata

def score_type_b(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
    Score TYPE_B evidence: DAG causal chains, ToC completeness.

    Expects:
    - elements: List of causal chain elements (up to 3)
    - completeness: DAG completeness score (0-1)

    Scoring:
    - Count causal elements (max 3)
    - Weight by completeness
    - Each element worth 1 point

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    completeness = evidence.get("completeness", ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L431_48", 0.0))

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_B: 'elements' must be a list")

    if not isinstance(completeness, (int, float)) or not (0 <= completeness <= 1):
        raise ModalityValidationError("TYPE_B: 'completeness' must be a number between 0 and 1")

    # Count valid elements (up to expected)
    element_count = min(len(elements), config.expected_elements or 3)

    # Calculate raw score: each element worth 1 point, weighted by completeness
    raw_score = float(element_count) * completeness

    # Clamp to valid range
    score = max(config.score_range[0], min(config.score_range[1], raw_score))

    metadata = {
        "element_count": element_count,
        "completeness": completeness,
        "raw_score": raw_score,
        "expected_elements": config.expected_elements,
    }

    logger.info(
        f"TYPE_B score: {score:.2f} "
        f"(elements={element_count}, completeness={completeness:.2f})"
    )

    return score, metadata

```

```
def score_type_c(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
    Score TYPE_C evidence: Coherence via inverted contradictions.

    Expects:
    - elements: List of coherence elements (up to 2)
    - coherence_score: Inverted contradiction score (0-1, higher is better)

    Scoring:
    - Count coherence elements (max 2)
    - Scale by coherence score
    - Scale to 0-3 range

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    coherence_score = evidence.get("coherence_score", ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L483_54", 0.0))

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_C: 'elements' must be a list")

    if not isinstance(coherence_score, (int, float)) or not (0 <= coherence_score <= 1):
        raise ModalityValidationError("TYPE_C: 'coherence_score' must be a number between 0 and 1")

    # Count valid elements (up to expected)
    element_count = min(len(elements), config.expected_elements or 2)

    # Calculate raw score: scale elements to range, weighted by coherence
    raw_score = (element_count / 2.0) * 3.0 * coherence_score

    # Clamp to valid range
    score = max(config.score_range[0], min(config.score_range[1], raw_score))

    metadata = {
        "element_count": element_count,
        "coherence_score": coherence_score,
        "raw_score": raw_score,
        "expected_elements": config.expected_elements,
    }

    logger.info(
        f"TYPE_C score: {score:.2f} "
        f"(elements={element_count}, coherence={coherence_score:.2f})"
    )

    return score, metadata

def score_type_d(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
    Score TYPE_D evidence: Pattern matching for baseline data.
```

```
    Expects:
    - elements: List of pattern matches (up to 3)
    - pattern_matches: Number of successful pattern matches

    Scoring:
    - Count pattern matches (max 3)
    - Weight by match quality if available
    - Scale to 0-3 range

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    pattern_matches = evidence.get("pattern_matches", 0)

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_D: 'elements' must be a list")

    if not isinstance(pattern_matches, (int, float)) or pattern_matches < 0:
        raise ModalityValidationError("TYPE_D: 'pattern_matches' must be a non-negative number")

    # Count valid elements (up to expected)
    element_count = min(len(elements), config.expected_elements or 3)

    # Use actual pattern matches if available, otherwise use element count
    match_count = min(pattern_matches, element_count) if pattern_matches > 0 else element_count

    # Calculate raw score: scale to 0-3 range
    raw_score = (match_count / 3.0) * 3.0

    # Clamp to valid range
    score = max(config.score_range[0], min(config.score_range[1], raw_score))

    metadata = {
        "element_count": element_count,
        "pattern_matches": match_count,
        "raw_score": raw_score,
        "expected_elements": config.expected_elements,
    }

    logger.info(
        f"TYPE_D score: {score:.2f} "
        f"(elements={element_count}, matches={match_count})"
    )

    return score, metadata

def score_type_e(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
    Score TYPE_E evidence: Financial budget traceability.

```

```
    Expects:
    - elements: List of budget elements
    - traceability: Boolean or numeric traceability score

    Scoring:
    - Boolean presence check
    - If numeric traceability provided, use that
    - Scale to 0-3 range

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    traceability = evidence.get("traceability", False)

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_E: 'elements' must be a list")

    # Handle both boolean and numeric traceability
    if isinstance(traceability, bool):
        traceability_score = ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L597_29", 1.0) if traceability else ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L597_54", 0.0)
    elif isinstance(traceability, (int, float)):
        if not (0 <= traceability <= 1):
            raise ModalityValidationError("TYPE_E: numeric 'traceability' must be between 0 and 1")
        traceability_score = float(traceability)
    else:
        raise ModalityValidationError("TYPE_E: 'traceability' must be boolean or numeric")

    # Count valid elements
    element_count = len(elements)
    has_elements = element_count > 0

    # Calculate raw score: presence check weighted by traceability
    raw_score = 3.0 * traceability_score if has_elements else ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L610_62", 0.0)

    # Clamp to valid range
    score = max(config.score_range[0], min(config.score_range[1], raw_score))

    metadata = {
        "element_count": element_count,
        "traceability": traceability_score,
        "raw_score": raw_score,
        "has_elements": has_elements,
    }

    logger.info(
        f"TYPE_E score: {score:.2f} "
        f"(elements={element_count}, traceability={traceability_score:.2f})"
    )

```

```
    return score, metadata

def score_type_f(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
    """
    Score TYPE_F evidence: Beach mechanism inference and plausibility.

    Expects:
    - elements: List of mechanism elements
    - plausibility: Plausibility score (0-1)

    Scoring:
    - Continuous scale based on plausibility
    - Weight by element presence
    - Scale to 0-3 range

    Args:
        evidence: Evidence dictionary
        config: Modality configuration

    Returns:
        Tuple of (score, metadata)
    """
    elements = evidence.get("elements", [])
    plausibility = evidence.get("plausibility", ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L650_48", 0.0))

    if not isinstance(elements, list):
        raise ModalityValidationError("TYPE_F: 'elements' must be a list")

    if not isinstance(plausibility, (int, float)) or not (0 <= plausibility <= 1):
        raise ModalityValidationError("TYPE_F: 'plausibility' must be a number between 0 and 1")

    # Count valid elements
    element_count = len(elements)

    # Calculate raw score: continuous scale weighted by plausibility
    raw_score = 3.0 * plausibility if element_count > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L662_61", 0.0)

    # Clamp to valid range
    score = max(config.score_range[0], min(config.score_range[1], raw_score))

    metadata = {
        "element_count": element_count,
        "plausibility": plausibility,
        "raw_score": raw_score,
    }

    logger.info(
        f"TYPE_F score: {score:.2f} "
        f"(elements={element_count}, plausibility={plausibility:.2f})"
    )

    return score, metadata

# Scoring function registry
```

```python
SCORING_FUNCTIONS = {
    ScoringModality.TYPE_A: score_type_a,
    ScoringModality.TYPE_B: score_type_b,
    ScoringModality.TYPE_C: score_type_c,
    ScoringModality.TYPE_D: score_type_d,
    ScoringModality.TYPE_E: score_type_e,
    ScoringModality.TYPE_F: score_type_f,
}

def determine_quality_level(
    normalized_score: float,
    thresholds: dict[str, float] | None = None,
) -> QualityLevel:
    """
    Determine quality level from normalized score.

    Args:
        normalized_score: Score normalized to 0-1 range
        thresholds: Optional custom thresholds

    Returns:
        Quality level

    Note:
        Default thresholds:
        - EXCELENTE: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L706_24", 0.85)
        - BUENO: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L707_20", 0.70)
        - ACEPTABLE: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L708_24", 0.55)
        - INSUFICIENTE: < ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L709_26", 0.55)
    """
    if thresholds is None:
        thresholds = {
            "EXCELENTE": ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L713_25", 0.85),
            "BUENO": ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L714_21", 0.70),
            "ACEPTABLE": ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L715_25", 0.55),
        }

    thresholds = _validate_quality_thresholds(thresholds)

    # Clamp score to account for minor floating-point drift
    normalized_score = clamp(float(normalized_score), ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L721_54", 0.0), ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L721_59", 1.0))

    if normalized_score >= thresholds["EXCELENTE"]:
        return QualityLevel.EXCELENTE
    elif normalized_score >= thresholds["BUENO"]:
        return QualityLevel.BUENO
    elif normalized_score >= thresholds["ACEPTABLE"]:
        return QualityLevel.ACEPTABLE
    else:
        return QualityLevel.INSUFICIENTE

def apply_scoring(
    question_global: int,
    base_slot: str,
    policy_area: str,
```

```
    dimension: str,
    evidence: dict[str, Any],
    modality: str,
    quality_thresholds: dict[str, float] | None = None,
) -> ScoredResult:
    """
    Apply scoring to evidence using specified modality.

    This is the main entry point for scoring. It:
    1. Validates evidence structure against modality
    2. Applies modality-specific scoring function
    3. Normalizes score to 0-1 range
    4. Determines quality level
    5. Returns reproducible ScoredResult

    Args:
        question_global: Global question number (1-300)
        base_slot: Question slot identifier
        policy_area: Policy area ID (PA01-PA10)
        dimension: Dimension ID (DIM01-DIM06)
        evidence: Evidence dictionary
        modality: Scoring modality (TYPE_A through TYPE_F)
        quality_thresholds: Optional custom quality thresholds

    Returns:
        ScoredResult

    Raises:
        ModalityValidationError: If evidence validation fails
        ScoringError: If scoring fails

    Note:
        This function has strict abortability. Any validation or scoring
        error will raise an exception and halt processing. No fallback
        or partial scoring is performed.
    """
    logger.info(
        f"Scoring question {question_global} ({base_slot}) "
        f"using {modality}"
    )

    # Parse modality
    try:
        modality_enum = ScoringModality(modality)
    except ValueError as e:
        raise ModalityValidationError(
            f"Invalid modality: {modality}. "
            f"Must be one of: {[m.value for m in ScoringModality]}"
        ) from e

    # Validate evidence structure
    ScoringValidator.validate(evidence, modality_enum)

    # Get modality configuration
    config = ScoringValidator.get_config(modality_enum)

```

```
    # Get scoring function
    scoring_func = SCORING_FUNCTIONS.get(modality_enum)
    if not scoring_func:
        raise ScoringError(f"No scoring function for {modality}")

    # Apply scoring
    try:
        score, metadata = scoring_func(evidence, config)
    except (ModalityValidationError, EvidenceStructureError, ScoringError) as e:
        logger.exception(f"Scoring failed for {modality}: {e}")
        raise ScoringError(f"Scoring failed for {modality}: {e}") from e
    except Exception as e:
        logger.exception(f"Unexpected error in scoring {modality}: {e}")
        raise ScoringError(f"Unexpected error in scoring {modality}: {e}") from e

    # Apply rounding
    rounded_score = apply_rounding(
        score,
        mode=config.rounding_mode,
        precision=config.rounding_precision,
    )

    min_score, max_score = config.score_range
    if max_score <= min_score:
        raise ScoringError(
            f"Invalid score range for {modality}: {config.score_range}"
        )

    # Guard against errant modality implementations
    clamped_score = clamp(rounded_score, min_score, max_score)
    score_clamped = not math.isclose(clamped_score, rounded_score, rel_tol=1e-9, abs_tol=1e-9)

    # Normalize score to 0-1 range
    normalized_score = (clamped_score - min_score) / (max_score - min_score)
    normalized_score = clamp(normalized_score, ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L826_47", 0.0), ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L826_52", 1.0))

    # Determine quality level
    quality_level = determine_quality_level(normalized_score, quality_thresholds)

    # Compute evidence hash for reproducibility
    evidence_hash = ScoredResult.compute_evidence_hash(evidence)

    # Build result
    result = ScoredResult(
        question_global=question_global,
        base_slot=base_slot,
        policy_area=policy_area,
        dimension=dimension,
        modality=modality,
        score=rounded_score,
        normalized_score=normalized_score,
        quality_level=quality_level.value,
        evidence_hash=evidence_hash,
        metadata={
            **metadata,
```

```
            "score_range": config.score_range,
            "rounding_mode": config.rounding_mode,
            "rounding_precision": config.rounding_precision,
            "score_clamped": score_clamped,
        },
    )

    logger.info(
        f"â\234\223 Scoring complete: score={rounded_score:.2f}, "
        f"normalized={normalized_score:.2f}, quality={quality_level.value}"
    )

    return result

__all__ = [
    "ScoringModality",
    "QualityLevel",
    "ScoringError",
    "ModalityValidationError",
    "EvidenceStructureError",
    "ScoredResult",
    "ModalityConfig",
    "ScoringValidator",
    "apply_scoring",
    "determine_quality_level",
]



================================================================================
FILE: src/farfan_pipeline/analysis/scoring.py
================================================================================

"""
SCORING MODULE - Question Scoring According to Questionnaire Monolith
=====================================================================
File: scoring.py
Code: SC
Purpose: Apply scoring modalities to question results

This module implements the scoring system for policy assessment questions.
All scoring modalities and quality thresholds are defined in the questionnaire
monolith specification (lines 34512-34607).

SCORING MODALITIES (6 types):
------------------------------
1. TYPE_A: Count 4 elements and scale to 0-3 (threshold=0.7 ratio)
   - Used when 4 specific policy elements must be present
   - Threshold: 70% of elements must be found to receive partial credit

2. TYPE_B: Count up to 3 elements, each worth 1 point
   - Used for independent policy components
   - Each element contributes equally to the final score

3. TYPE_C: Count 2 elements and scale to 0-3 (threshold=0.5 ratio)
   - Used when 2 critical policy elements must be present
```

```
   - Threshold: 50% of elements must be found to receive partial credit

4. TYPE_D: Count 3 elements, weighted [0.4, 0.3, 0.3]
   - Used when policy elements have different importance
   - First element has highest weight (40%), others equal (30% each)

5. TYPE_E: Boolean presence check
   - Binary scoring: element is present (3 points) or absent (0 points)

6. TYPE_F: Semantic matching with cosine similarity (normalized_continuous)
   - Uses text similarity to assess policy alignment
   - Continuous score based on semantic similarity (0.0-1.0 range)

QUALITY LEVELS:
---------------
Quality levels are determined from normalized scores (0.0-1.0 scale):
- EXCELLENT: â\211¥ 0.85 (85th percentile) - green indicator
- GOOD: â\211¥ 0.70 (70th percentile) - blue indicator
- ACCEPTABLE: â\211¥ 0.55 (55th percentile) - yellow indicator
- INSUFFICIENT: < 0.55 (below 55th percentile) - red indicator

CORE METHODS:
-------------
1. MicroQuestionScorer.score_type_a() - TYPE_A scoring logic
2. MicroQuestionScorer.score_type_b() - TYPE_B scoring logic
3. MicroQuestionScorer.score_type_c() - TYPE_C scoring logic
4. MicroQuestionScorer.score_type_d() - TYPE_D scoring logic
5. MicroQuestionScorer.score_type_e() - TYPE_E scoring logic
6. MicroQuestionScorer.score_type_f() - TYPE_F scoring logic
7. MicroQuestionScorer.apply_scoring_modality() - Dispatcher for modalities
8. MicroQuestionScorer.determine_quality_level() - Maps scores to quality levels

DATA FLOW:
----------
Input: QuestionResult with evidence from Phase 2 evaluation
Output: ScoredResult with score (0-3 range) and quality level classification

REFERENCE:
----------
Questionnaire monolith specification lines 34512-34607
"""

import logging
from dataclasses import dataclass, field
from enum import Enum
from typing import Any

import numpy as np
from farfan_pipeline.core.parameters import ParameterLoaderV2
from farfan_pipeline.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

# ============================================================================
# ENUMS - EXACTOS DEL MONOLITH
# ============================================================================
```

```

class ScoringModality(Enum):
    """Modalidades de scoring del monolith (lÃ-nea 34535)."""
    TYPE_A = "TYPE_A"  # Count 4 elements and scale to 0-3
    TYPE_B = "TYPE_B"  # Count up to 3 elements, each worth 1 point
    TYPE_C = "TYPE_C"  # Count 2 elements and scale to 0-3
    TYPE_D = "TYPE_D"  # Count 3 elements, weighted
    TYPE_E = "TYPE_E"  # Boolean presence check
    TYPE_F = "TYPE_F"  # Semantic matching with cosine similarity

class QualityLevel(Enum):
    """Niveles de calidad micro (lÃ-nea 34513)."""
    EXCELENTE = "EXCELENTE"    # â\211¥ 0.85
    BUENO = "BUENO"           # â\211¥ 0.70
    ACEPTABLE = "ACEPTABLE"   # â\211¥ 0.55
    INSUFICIENTE = "INSUFICIENTE"  # < 0.55

# ============================================================================
# DATACLASSES
# ============================================================================

@dataclass
class ScoringConfig:
    """
    Scoring configuration extracted from questionnaire monolith specification.

    This configuration defines all parameters for the six scoring modalities
    and quality level thresholds. All values are derived from the questionnaire
    monolith specification (lines 34512-34607).

    Attributes:
        TYPE_A Configuration (line 34568):
            type_a_threshold: Ratio threshold for partial credit (0.0-1.0 scale, default: 0.7)
                             Elements found / expected must exceed this to receive credit
            type_a_max_score: Maximum score achievable (default: 3.0 points)
            type_a_expected_elements: Number of elements expected (default: 4 elements)

        TYPE_B Configuration (line 34574):
            type_b_max_score: Maximum score achievable (default: 3.0 points)
            type_b_max_elements: Maximum elements to count (default: 3 elements)

        TYPE_C Configuration (line 34580):
            type_c_threshold: Ratio threshold for partial credit (0.0-1.0 scale, default: 0.5)
                             Elements found / expected must exceed this to receive credit
            type_c_max_score: Maximum score achievable (default: 3.0 points)
            type_c_expected_elements: Number of elements expected (default: 2 elements)

        TYPE_D Configuration (line 34586):
            type_d_weights: Importance weights for each element (0.0-1.0 scale per weight,
                           must sum to 1.0, default: [0.4, 0.3, 0.3])
                           First element weighted 40%, second and third 30% each
            type_d_max_score: Maximum score achievable (default: 3.0 points)
            type_d_expected_elements: Number of elements expected (default: 3 elements)

        TYPE_E Configuration (line 34596):
            type_e_max_score: Maximum score achievable (default: 3.0 points)
```

```
                             Binary: full score if present, 0 if absent

        TYPE_F Configuration (line 34601):
            type_f_max_score: Maximum score achievable (default: 3.0 points)
            type_f_normalization: Normalization method for similarity scores (default: "minmax")
                                 Options: "minmax", "zscore", "none"

        Quality Level Thresholds (line 34513):
            level_excelente_min: Minimum normalized score for EXCELLENT (0.0-1.0 scale, default: 0.85)
            level_bueno_min: Minimum normalized score for GOOD (0.0-1.0 scale, default: 0.70)
            level_aceptable_min: Minimum normalized score for ACCEPTABLE (0.0-1.0 scale, default: 0.55)
            level_insuficiente_min: Minimum normalized score for INSUFFICIENT (0.0-1.0 scale, default: 0.0)
    """

    # TYPE_A config (line 34568)
    type_a_threshold: float = 0.7  # Ratio (0.0-1.0): proportion of elements required
    type_a_max_score: float = 3.0  # Points: maximum achievable score
    type_a_expected_elements: int = 4  # Count: number of policy elements to check

    # TYPE_B config (line 34574)
    type_b_max_score: float = 3.0  # Points: maximum achievable score
    type_b_max_elements: int = 3  # Count: maximum elements to score

    # TYPE_C config (line 34580)
    type_c_threshold: float = 0.5  # Ratio (0.0-1.0): proportion of elements required
    type_c_max_score: float = 3.0  # Points: maximum achievable score
    type_c_expected_elements: int = 2  # Count: number of policy elements to check

    # TYPE_D config (line 34586)
    type_d_weights: list[float] = field(default_factory=lambda: [0.4, 0.3, 0.3])  # Weights (sum to 1.0): element importance
    type_d_max_score: float = 3.0  # Points: maximum achievable score
    type_d_expected_elements: int = 3  # Count: number of policy elements to check

    # TYPE_E config (line 34596)
    type_e_max_score: float = 3.0  # Points: maximum achievable score (binary: 3.0 or 0.0)

    # TYPE_F config (line 34601)
    type_f_max_score: float = 3.0  # Points: maximum achievable score
    type_f_normalization: str = "minmax"  # Method: "minmax", "zscore", or "none"

    # Quality levels (line 34513) - All thresholds are normalized scores (0.0-1.0 scale)
    level_excelente_min: float = 0.85  # Ratio (0.0-1.0): minimum for EXCELLENT quality
    level_bueno_min: float = 0.70  # Ratio (0.0-1.0): minimum for GOOD quality
    level_aceptable_min: float = 0.55  # Ratio (0.0-1.0): minimum for ACCEPTABLE quality
    level_insuficiente_min: float = 0.0  # Ratio (0.0-1.0): minimum for INSUFFICIENT quality

@dataclass
class Evidence:
    """
    Evidencia extraÃ-da para una pregunta.
    Producida por evaluadores en FASE 2.
    """
    elements_found: list[str] = field(default_factory=list)
    confidence_scores: list[float] = field(default_factory=list)
    semantic_similarity: float | None = None
    pattern_matches: dict[str, int] = field(default_factory=dict)
```

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

@dataclass
class ScoredResult:
    """
    Resultado con score aplicado.
    Output de este mÃ³dulo.
    """
    question_id: str
    question_global: int
    scoring_modality: ScoringModality
    raw_score: float  # 0-3
    normalized_score: float  # 0-1 (raw_score / 3.0)
    quality_level: QualityLevel
    quality_color: str  # "green", "blue", "yellow", "red"
    evidence: Evidence
    scoring_details: dict[str, Any] = field(default_factory=dict)

# ============================================================================
# CLASE: MicroQuestionScorer
# ============================================================================

class MicroQuestionScorer:
    """
    Aplicador de modalidades de scoring segÃºn monolith.

    Responsabilidades:
    - Aplicar TYPE_A, TYPE_B, TYPE_C, TYPE_D, TYPE_E, TYPE_F
    - Calcular score 0-3
    - Determinar nivel de calidad (EXCELENTE/BUENO/ACEPTABLE/INSUFICIENTE)
    """

    def __init__(self, config: ScoringConfig | None = None) -> None:
        """
        Inicializa scorer con configuraciÃ³n del monolith.

        Args:
            config: ConfiguraciÃ³n de scoring (defaults del monolith si None)
        """
        self.config = config or ScoringConfig()
        self.logger = logger

    # ========================================================================
    # MÃ\211TODO 1: SCORE TYPE_A
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a")
    def score_type_a(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 1: TYPE_A - Count 4 elements and scale to 0-3.

        ESPECIFICACIÃ\223N (lÃ-nea 34568 del monolith):
        - Aggregation: "presence_threshold"
        - Threshold: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L245_21", 0.7)
        - Max_score: 3
        - Expected_elements: 4
```

```

        LÃ\223GICA:
        1. Contar elementos encontrados (expected: 4)
        2. Calcular ratio = found / 4
        3. Si ratio >= ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L252_23", 0.7): aplicar escala proporcional
        4. Si ratio < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L253_22", 0.7): penalizar fuertemente

        ESCALA:
        - 4/4 elementos (100%) â\206\222 3.0
        - 3/4 elementos (75%) â\206\222 2.25
        - 2/4 elementos (50%) â\206\222 penalizado
        - 1/4 elementos (25%) â\206\222 penalizado
        - 0/4 elementos (0%) â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L260_31", 0.0)

        Args:
            evidence: Evidencia extraÃda con elements_found

        Returns:
            Tuple de (score, details)
        """
        elements_found = len(evidence.elements_found)
        expected = self.config.type_a_expected_elements
        threshold = self.config.type_a_threshold
        max_score = self.config.type_a_max_score

        # Calcular ratio
        ratio = elements_found / expected if expected > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L274_63", 0.0)

        # Aplicar threshold del monolith
        if ratio >= threshold:
            # Escala proporcional: ratio * max_score
            score = ratio * max_score
        else:
            # PenalizaciÃ³n: escala cuadrÃ¡tica para ratios bajos
            score = (ratio / threshold) * (ratio * max_score)

        # Clip al rango [0, max_score]
        score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L285_20", 0.0), min(max_score, score))

        details = {
            'modality': 'TYPE_A',
            'elements_found': elements_found,
            'expected_elements': expected,
            'ratio': ratio,
            'threshold': threshold,
            'threshold_met': ratio >= threshold,
            'raw_score': score,
            'formula': 'ratio * max_score if ratio >= threshold else penalized'
        }

        self.logger.debug(f"TYPE_A: {elements_found}/{expected} elementos ({ratio:.2f}) â\206\222 score={score:.2f}")

```

```
        return score, details

    # ========================================================================
    # MÃ\211TODO 2: SCORE TYPE_B
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b")
    def score_type_b(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 2: TYPE_B - Count up to 3 elements, each worth 1 point.

        ESPECIFICACIÃ\223N (lÃnea 34574 del monolith):
        - Aggregation: "binary_sum"
        - Max_score: 3
        - Max_elements: 3

        LÃ\223GICA:
        1. Contar elementos encontrados (max: 3)
        2. Cada elemento = 1 punto
        3. Score = min(elements_found, 3)

        ESCALA:
        - 3+ elementos â\206\222 3.0
        - 2 elementos â\206\222 2.0
        - 1 elemento â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b", "auto_param_L324_23", 1.0)
        - 0 elementos â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b", "auto_param_L325_24", 0.0)

        Args:
            evidence: Evidencia extraÃda con elements_found

        Returns:
            Tuple de (score, details)
        """
        elements_found = len(evidence.elements_found)
        max_elements = self.config.type_b_max_elements
        max_score = self.config.type_b_max_score

        # Binary sum: cada elemento vale 1 punto, hasta max_elements
        score = min(float(elements_found), max_elements)

        # Asegurar que no excede max_score
        score = min(score, max_score)

        details = {
            'modality': 'TYPE_B',
            'elements_found': elements_found,
            'max_elements': max_elements,
            'raw_score': score,
            'formula': 'min(elements_found, 3)'
        }

        self.logger.debug(f"TYPE_B: {elements_found} elementos â\206\222 score={score:.2f}")

        return score, details

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

```
    # MÃ\211TODO 3: SCORE TYPE_C
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c")
    def score_type_c(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 3: TYPE_C - Count 2 elements and scale to 0-3.

        ESPECIFICACIÃ\223N (lÃ-nea 34580 del monolith):
        - Aggregation: "presence_threshold"
        - Threshold: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L366_21", 0.5)
        - Max_score: 3
        - Expected_elements: 2

        LÃ\223GICA:
        1. Contar elementos encontrados (expected: 2)
        2. Calcular ratio = found / 2
        3. Si ratio >= ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L373_23", 0.5): aplicar escala proporcional
        4. Si ratio < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L374_22", 0.5): penalizar

        ESCALA:
        - 2/2 elementos (100%) â\206\222 3.0
        - 1/2 elementos (50%) â\206\222 1.5
        - 0/2 elementos (0%) â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L379_31", 0.0)

        Args:
            evidence: Evidencia extraÃ-da con elements_found

        Returns:
            Tuple de (score, details)
        """
        elements_found = len(evidence.elements_found)
        expected = self.config.type_c_expected_elements
        threshold = self.config.type_c_threshold
        max_score = self.config.type_c_max_score

        # Calcular ratio
        ratio = elements_found / expected if expected > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L393_63", 0.0)

        # Aplicar threshold del monolith
        if ratio >= threshold:
            # Escala proporcional
            score = ratio * max_score
        else:
            # PenalizaciÃ³n cuadrÃ¡tica
            score = (ratio / threshold) * (ratio * max_score)

        # Clip al rango [0, max_score]
        score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L404_20", 0.0), min(max_score, score))

        details = {
            'modality': 'TYPE_C',
            'elements_found': elements_found,
```

```
            'expected_elements': expected,
            'ratio': ratio,
            'threshold': threshold,
            'threshold_met': ratio >= threshold,
            'raw_score': score,
            'formula': 'ratio * max_score if ratio >= threshold else penalized'
        }

        self.logger.debug(f"TYPE_C: {elements_found}/{expected} elementos ({ratio:.2f}) â\206\222 score={score:.2f}")

        return score, details

    # ========================================================================
    # MÃ\211TODO 4: SCORE TYPE_D
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d")
    def score_type_d(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 4: TYPE_D - Count 3 elements, weighted [ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_55", 0.4), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_60", 0.3), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_65", 0.3)].

        ESPECIFICACIÃ\223N (lÃ-nea 34586 del monolith):
        - Aggregation: "weighted_sum"
        - Weights: [ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_20", 0.4), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_25", 0.3), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_30", 0.3)]
        - Max_score: 3
        - Expected_elements: 3

        LÃ\223GICA:
        1. Se esperan 3 elementos con importancia diferente
        2. Elemento 1: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L438_28", 0.4) (mÃ¡s importante)
        3. Elemento 2: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L439_28", 0.3)
        4. Elemento 3: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L440_28", 0.3)
        5. Score = (sum of weights for found elements) * max_score

        ESCALA:
        - 3 elementos (todos) â\206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L444_44", 1.0) â\206\222 3.0
        - 2 elementos (ej: elem1+elem2) â\206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L445_54", 0.7) â\206\222 2.1
        - 1 elemento (ej: elem1) â\206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L446_47", 0.4) â\206\222 1.2
        - 0 elementos â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L447_24", 0.0)

        Args:
            evidence: Evidencia extraÃ-da con elements_found y confidence_scores

        Returns:
            Tuple de (score, details)
        """
        elements_found = len(evidence.elements_found)
        expected = self.config.type_d_expected_elements
```

```
        weights = self.config.type_d_weights
        max_score = self.config.type_d_max_score

        # Calcular suma ponderada
        # Asumimos que elements_found estÃ¡ ordenado por importancia
        # o usamos confidence_scores si estÃ¡n disponibles
        if evidence.confidence_scores and len(evidence.confidence_scores) >= elements_found:
            # Ordenar por confidence (descendente) y aplicar pesos
            sorted_confidences = sorted(evidence.confidence_scores[:elements_found], reverse=True)
            weighted_sum = sum(
                conf * weights[i]
                for i, conf in enumerate(sorted_confidences)
                if i < len(weights)
            )
        else:
            # Sin confidence scores: asumir presencia binaria
            weighted_sum = sum(weights[:min(elements_found, len(weights))])

        # Score = weighted_sum * max_score
        score = weighted_sum * max_score

        # Clip al rango [0, max_score]
        score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L479_20", 0.0), min(max_score, score)
)


        details = {
            'modality': 'TYPE_D',
            'elements_found': elements_found,
            'expected_elements': expected,
            'weights': weights,
            'weighted_sum': weighted_sum,
            'raw_score': score,
            'formula': 'weighted_sum * max_score'
        }

        self.logger.debug(f"TYPE_D: {elements_found}/{expected} elementos, weighted_sum={weighted_sum:.2f} â\206\222 score={score:.2f}")

        return score, details

    # ========================================================================
    # MÃ\211TODO 5: SCORE TYPE_E
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e")
    def score_type_e(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 5: TYPE_E - Boolean presence check.

        ESPECIFICACIÃ\223N (lÃnea 34596 del monolith):
        - Aggregation: "binary_presence"
        - Max_score: 3

        LÃ\223GICA:
        1. Verificar si existe evidencia (binario: sÃ/no)
        2. Si existe: 3.0
        3. Si no existe: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L511_25", 0.0)
```

```

        ESCALA:
        - Evidencia presente â\206\222 3.0
        - Evidencia ausente â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L515_30", 0.0)

        Args:
            evidence: Evidencia extraÃ-da

        Returns:
            Tuple de (score, details)
        """
        max_score = self.config.type_e_max_score

        # Verificar presencia de cualquier evidencia
        has_evidence = (
            len(evidence.elements_found) > 0 or
            bool(evidence.pattern_matches) or
            (evidence.semantic_similarity is not None and evidence.semantic_similarity > ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L529_89", 0.5))
        )

        # Binary: todo o nada
        score = max_score if has_evidence else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L533_47", 0.0)

        details = {
            'modality': 'TYPE_E',
            'has_evidence': has_evidence,
            'elements_found': len(evidence.elements_found),
            'pattern_matches': len(evidence.pattern_matches),
            'semantic_similarity': evidence.semantic_similarity,
            'raw_score': score,
            'formula': 'max_score if has_evidence else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L542_55", 0.0)'
        }

        self.logger.debug(f"TYPE_E: evidencia={'presente' if has_evidence else 'ausente'} â\206\222 score={score:.2f}")

        return score, details

    # ========================================================================
    # MÃ\211TODO 6: SCORE TYPE_F
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f")
    def score_type_f(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÃ\211TODO 6: TYPE_F - Semantic matching with cosine similarity.

        ESPECIFICACIÃ\223N (lÃ-nea 34601 del monolith):
        - Aggregation: "normalized_continuous"
        - Normalization: "minmax"
        - Max_score: 3

        LÃ\223GICA:
        1. Usar semantic_similarity (rango 0-1)
```

```
        2. Normalizar con minmax
        3. Score = normalized_similarity * max_score

        ESCALA:
        - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L569_23", 1.0) â\206\222 3.0
        - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L570_23", 0.75) â\206\222 2.25
        - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L571_23", 0.5) â\206\222 1.5
        - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L572_23", 0.25) â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L572_30", 0.75)
        - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L573_23", 0.0) â\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L573_29", 0.0)

        Args:
            evidence: Evidencia con semantic_similarity

        Returns:
            Tuple de (score, details)
        """
        max_score = self.config.type_f_max_score

        # Obtener similarity
        if evidence.semantic_similarity is not None:
            similarity = evidence.semantic_similarity
        # Fallback: calcular promedio de confidence_scores
        elif evidence.confidence_scores:
            similarity = float(np.mean(evidence.confidence_scores))
        else:
            similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "similarity", 0.0) # Refactored

        # NormalizaciÃ³n minmax (ya estÃ¡ en rango 0-1)
        normalized_similarity = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L593_36", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L593_45", 1.0), similarity))

        # Score continuo
        score = normalized_similarity * max_score

        details = {
            'modality': 'TYPE_F',
            'semantic_similarity': similarity,
            'normalized_similarity': normalized_similarity,
            'raw_score': score,
            'formula': 'normalized_similarity * max_score'
        }

        self.logger.debug(f"TYPE_F: similarity={similarity:.3f} â\206\222 score={score:.2f}")

        return score, details

    # ========================================================================
    # MÃ\211TODO 7: APPLY SCORING MODALITY (ORQUESTADOR)
    # ========================================================================

    def apply_scoring_modality(
        self,
        question_id: str,
        question_global: int,
```

```
        modality: ScoringModality,
        evidence: Evidence
    ) -> ScoredResult:
        """
        MÃ\211TODO 7: Aplica la modalidad de scoring correspondiente.

        ORQUESTADOR que delega a mÃ©todos 1-6 segÃºn modality.

        Args:
            question_id: ID de pregunta (ej: "Q001")
            question_global: NÃºmero global (1-305)
            modality: Modalidad de scoring
            evidence: Evidencia extraÃda

        Returns:
            ScoredResult con score 0-3 y nivel de calidad
        """
        self.logger.info(f"Aplicando scoring {modality.value} a {question_id}")

        # Delegar a mÃ©todo especÃfico
        if modality == ScoringModality.TYPE_A:
            raw_score, details = self.score_type_a(evidence)

        elif modality == ScoringModality.TYPE_B:
            raw_score, details = self.score_type_b(evidence)

        elif modality == ScoringModality.TYPE_C:
            raw_score, details = self.score_type_c(evidence)

        elif modality == ScoringModality.TYPE_D:
            raw_score, details = self.score_type_d(evidence)

        elif modality == ScoringModality.TYPE_E:
            raw_score, details = self.score_type_e(evidence)

        elif modality == ScoringModality.TYPE_F:
            raw_score, details = self.score_type_f(evidence)

        else:
            raise ValueError(f"Modalidad desconocida: {modality}")

        # Normalizar a 0-1
        normalized_score = raw_score / 3.0

        # Determinar nivel de calidad
        quality_level, quality_color = self.determine_quality_level(normalized_score)

        # Construir resultado
        scored_result = ScoredResult(
            question_id=question_id,
            question_global=question_global,
            scoring_modality=modality,
            raw_score=raw_score,
            normalized_score=normalized_score,
            quality_level=quality_level,
            quality_color=quality_color,
```

```
            evidence=evidence,
            scoring_details=details
        )

        self.logger.info(
            f"â\234\223 {question_id}: score={raw_score:.2f}/3.0 "
            f"({normalized_score:.2%}), nivel={quality_level.value}"
        )

        return scored_result

    # ========================================================================
    # MÃ\211TODO 8: DETERMINE QUALITY LEVEL
    # ========================================================================

    @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level")
    def determine_quality_level(self, normalized_score: float) -> tuple[QualityLevel, str]:
        """
        MÃ\211TODO 8: Determina nivel de calidad segÃºn umbrales del monolith.

        UMBRALES (lÃnea 34513 del monolith):
        - EXCELENTE: â\211¥ ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L695_23", 0.85) (verde)
        - BUENO: â\211¥ ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L696_19", 0.70) (azul)
        - ACEPTABLE: â\211¥ ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L697_23", 0.55) (amarillo)
        - INSUFICIENTE: < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L698_26", 0.55) (rojo)

        Args:
            normalized_score: Score en rango 0-1

        Returns:
            Tuple de (QualityLevel, color)
        """
        if normalized_score >= self.config.level_excelente_min:
            return QualityLevel.EXCELENTE, "green"

        elif normalized_score >= self.config.level_bueno_min:
            return QualityLevel.BUENO, "blue"

        elif normalized_score >= self.config.level_aceptable_min:
            return QualityLevel.ACEPTABLE, "yellow"

        else:
            return QualityLevel.INSUFICIENTE, "red"

# ========================================================================
# FUNCIÃ\223N DE CONVENIENCIA
# ========================================================================

def score_question(
    question_id: str,
    question_global: int,
    modality_str: str,
    evidence_dict: dict[str, Any]
```

```
) -> ScoredResult:
    """
    FunciÃ³n de conveniencia para scoring de una pregunta.

    Args:
        question_id: ID de pregunta
        question_global: NÃºmero global
        modality_str: String de modalidad ("TYPE_A", "TYPE_B", etc.)
        evidence_dict: Diccionario con evidencia

    Returns:
        ScoredResult
    """
    # Parsear modalidad
    modality = ScoringModality(modality_str)

    # Construir Evidence
    evidence = Evidence(
        elements_found=evidence_dict.get('elements_found', []),
        confidence_scores=evidence_dict.get('confidence_scores', []),
        semantic_similarity=evidence_dict.get('semantic_similarity'),
        pattern_matches=evidence_dict.get('pattern_matches', {}),
        metadata=evidence_dict.get('metadata', {})
    )

    # Aplicar scoring
    scorer = MicroQuestionScorer()
    result = scorer.apply_scoring_modality(
        question_id=question_id,
        question_global=question_global,
        modality=modality,
        evidence=evidence
    )

    return result

# ============================================================================
# EJEMPLO DE USO
# ============================================================================

# Note: Main entry point and examples removed to maintain I/O boundary separation.
    print("="*80)



================================================================================
FILE: src/farfan_pipeline/analysis/spc_causal_bridge.py
================================================================================

"""
SPC to TeoriaCambio Bridge - Causal Graph Construction.

This module bridges Smart Policy Chunks (SPC) chunk graphs to causal DAG
representations for integration with TeoriaCambio (Theory of Change) analysis.
"""

```

```python
from __future__ import annotations

import logging
from typing import Any

from farfan_pipeline.core.calibration.decorators import calibrated_method
from farfan_pipeline.core.parameters import ParameterLoaderV2

try:
    import networkx as nx

    HAS_NETWORKX = True
except ImportError:
    HAS_NETWORKX = False
    nx = None  # type: ignore

try:
    from farfan_pipeline.processing.models import CanonPolicyPackage, ChunkGraph

    HAS_CPP_MODELS = True
except ImportError:
    HAS_CPP_MODELS = False
    CanonPolicyPackage = None  # type: ignore
    ChunkGraph = None  # type: ignore

logger = logging.getLogger(__name__)


class SPCCausalBridge:
    """
    Converts SPC chunk graph to causal DAG for Theory of Change analysis.

    This bridge enables causal analysis by mapping semantic chunk relationships
    (sequential, hierarchical, reference, dependency) to causal weights that
    can be used by downstream causal inference methods.
    """

    # Mapping of SPC edge types to causal weights
    # Higher weight = stronger causal relationship
    CAUSAL_WEIGHTS: dict[str, float] = {
        "sequential": 0.3,  # Weak temporal causality (A then B)
        "hierarchical": 0.7,  # Strong structural causality (A contains/governs B)
        "reference": 0.5,  # Medium evidential causality (A references B)
        "dependency": 0.9,  # Strong logical causality (A requires B)
    }

    def __init__(self) -> None:
        """Initialize the SPC causal bridge."""
        if not HAS_NETWORKX:
            logger.warning(
                "NetworkX not available. SPCCausalBridge will have limited functionality. "
                "Install networkx for full causal graph construction."
            )
        if not HAS_CPP_MODELS:
            logger.warning(
                "CPP models not available. build_causal_graph_from_cpp will have limited functionality."
```

```
            )

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge.build_causal_graph_from_cpp"
    )
    def build_causal_graph_from_cpp(self, cpp: Any) -> Any:
        """
        Convert CanonPolicyPackage to causal DAG via chunk_graph extraction.

        This method serves as the Phase 1-2 adapter integration point, enabling
        D4/D6 executors to perform Theory of Change analysis on the rich
        semantic structure preserved in CanonPolicyPackage.

        Args:
            cpp: CanonPolicyPackage from Phase 1 ingestion

        Returns:
            NetworkX DiGraph representing causal relationships, or None if NetworkX unavailable

        Raises:
            ValueError: If cpp is invalid or missing chunk_graph
            ImportError: If required models not available
        """
        if not HAS_NETWORKX:
            logger.error("NetworkX required for causal graph construction")
            return None

        if not HAS_CPP_MODELS:
            logger.error("CPP models not available for processing")
            return None

        if not cpp:
            raise ValueError("cpp (CanonPolicyPackage) cannot be None or empty")

        if not hasattr(cpp, "chunk_graph") or not cpp.chunk_graph:
            raise ValueError(
                "CanonPolicyPackage must have a valid chunk_graph attribute. "
                "Ensure Phase 1 ingestion completed successfully."
            )

        chunk_graph = cpp.chunk_graph

        if not isinstance(chunk_graph, ChunkGraph):
            raise ValueError(
                f"Expected ChunkGraph instance, got {type(chunk_graph).__name__}"
            )

        if not chunk_graph.chunks:
            logger.warning("ChunkGraph has no chunks, returning empty causal graph")
            return nx.DiGraph()

        logger.info(
            f"Extracting causal graph from CanonPolicyPackage: "
            f"{len(chunk_graph.chunks)} chunks, {len(chunk_graph.edges)} edges"
        )

```

```
        chunk_graph_dict = self._convert_chunk_graph_to_dict(chunk_graph)

        causal_graph = self.build_causal_graph_from_spc(chunk_graph_dict)

        if causal_graph is not None:
            self._enhance_graph_with_cpp_metadata(causal_graph, cpp)

        return causal_graph

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._convert_chunk_graph_to_dict"
    )
    def _convert_chunk_graph_to_dict(self, chunk_graph: Any) -> dict:
        """
        Convert ChunkGraph object to dictionary format for SPC processing.

        Extracts nodes and edges from the ChunkGraph dataclass structure
        and converts them to the dictionary format expected by build_causal_graph_from_spc.

        Args:
            chunk_graph: ChunkGraph instance from CanonPolicyPackage

        Returns:
            Dictionary with 'nodes' and 'edges' keys containing graph structure
        """
        nodes = []
        for chunk_id, chunk in chunk_graph.chunks.items():
            node = {
                "id": chunk_id,
                "type": (
                    getattr(chunk, "resolution", "MESO").name
                    if hasattr(getattr(chunk, "resolution", None), "name")
                    else "MESO"
                ),
                "text": getattr(chunk, "text", "")[:200],
                "confidence": getattr(
                    getattr(chunk, "confidence", None), "layout", 1.0
                ),
            }

            if hasattr(chunk, "policy_area_id") and chunk.policy_area_id:
                node["policy_area_id"] = chunk.policy_area_id
            if hasattr(chunk, "dimension_id") and chunk.dimension_id:
                node["dimension_id"] = chunk.dimension_id

            nodes.append(node)

        edges = []
        for edge_tuple in chunk_graph.edges:
            if len(edge_tuple) >= 3:
                source, target, relation_type = (
                    edge_tuple[0],
                    edge_tuple[1],
                    edge_tuple[2],
                )
            elif len(edge_tuple) == 2:
```

```
                source, target = edge_tuple[0], edge_tuple[1]
                relation_type = "sequential"
            else:
                logger.warning(f"Malformed edge tuple: {edge_tuple}, skipping")
                continue

            edges.append(
                {
                    "source": source,
                    "target": target,
                    "type": self._normalize_edge_type(relation_type),
                }
            )

        logger.debug(f"Converted ChunkGraph: {len(nodes)} nodes, {len(edges)} edges")

        return {"nodes": nodes, "edges": edges}

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._normalize_edge_type"
    )
    def _normalize_edge_type(self, relation_type: str) -> str:
        """
        Normalize edge relation types to causal weights mapping.

        Args:
            relation_type: Original relation type from ChunkGraph

        Returns:
            Normalized edge type compatible with CAUSAL_WEIGHTS
        """
        relation_lower = str(relation_type).lower().strip()

        if relation_lower in self.CAUSAL_WEIGHTS:
            return relation_lower

        type_mapping = {
            "seq": "sequential",
            "sequence": "sequential",
            "hier": "hierarchical",
            "hierarchy": "hierarchical",
            "parent": "hierarchical",
            "child": "hierarchical",
            "ref": "reference",
            "cite": "reference",
            "citation": "reference",
            "dep": "dependency",
            "require": "dependency",
            "requires": "dependency",
            "prerequisite": "dependency",
        }

        return type_mapping.get(relation_lower, "sequential")

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._enhance_graph_with_cpp_metadata"
```

```
    )
    def _enhance_graph_with_cpp_metadata(self, G: Any, cpp: Any) -> None:
        """
        Enhance causal graph with CanonPolicyPackage metadata.

        Enriches nodes with quality metrics, policy manifest data, and provenance
        information from the CPP, enabling richer Theory of Change analysis.

        Args:
            G: NetworkX DiGraph to enhance
            cpp: CanonPolicyPackage with metadata
        """
        if not HAS_NETWORKX or G is None:
            return

        G.graph["schema_version"] = getattr(cpp, "schema_version", "unknown")

        if hasattr(cpp, "quality_metrics") and cpp.quality_metrics:
            qm = cpp.quality_metrics
            G.graph["quality_metrics"] = {
                "provenance_completeness": getattr(qm, "provenance_completeness", 0.0),
                "structural_consistency": getattr(qm, "structural_consistency", 0.0),
                "boundary_f1": getattr(qm, "boundary_f1", 0.0),
                "kpi_linkage_rate": getattr(qm, "kpi_linkage_rate", 0.0),
                "budget_consistency_score": getattr(
                    qm, "budget_consistency_score", 0.0
                ),
            }

        if hasattr(cpp, "policy_manifest") and cpp.policy_manifest:
            pm = cpp.policy_manifest
            G.graph["policy_manifest"] = {
                "axes": getattr(pm, "axes", []),
                "programs": getattr(pm, "programs", []),
                "projects": getattr(pm, "projects", []),
                "years": getattr(pm, "years", []),
                "territories": getattr(pm, "territories", []),
            }

        logger.debug(
            f"Enhanced causal graph with CPP metadata: {len(G.graph)} graph attributes"
        )

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge.build_causal_graph_from_spc"
    )
    def build_causal_graph_from_spc(self, chunk_graph: dict) -> Any:
        """
        Convert SPC chunk graph to causal DAG.

        Args:
            chunk_graph: Dictionary with 'nodes' and 'edges' from chunk graph

        Returns:
            NetworkX DiGraph representing causal relationships, or None if NetworkX unavailable

```

```
        Raises:
            ValueError: If chunk_graph is invalid
        """
        if not HAS_NETWORKX:
            logger.error("NetworkX required for causal graph construction")
            return None

        if not chunk_graph or not isinstance(chunk_graph, dict):
            raise ValueError("chunk_graph must be a non-empty dictionary")

        nodes = chunk_graph.get("nodes", [])
        edges = chunk_graph.get("edges", [])

        if not nodes:
            logger.warning("No nodes in chunk graph, returning empty graph")
            return nx.DiGraph()

        # Create directed graph
        G = nx.DiGraph()

        # Add nodes with attributes
        for node in nodes:
            node_id = node.get("id")
            if node_id is None:
                continue

            G.add_node(
                f"chunk_{node_id}",
                chunk_type=node.get("type", "unknown"),
                text_summary=node.get("text", "")[:100],  # First 100 chars
                confidence=node.get(
                    "confidence",
                    ParameterLoaderV2.get(
                        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge.build_causal_graph_from_spc",
                        "auto_param_L91_50",
                        0.0,
                    ),
                ),
            )

        # Add edges with causal interpretation
        for edge in edges:
            source = edge.get("source")
            target = edge.get("target")
            edge_type = edge.get("type", "sequential")

            if source is None or target is None:
                continue

            # Convert to node IDs
            # Handle both string and integer IDs
            if (
                isinstance(source, str)
                and not source.startswith("chunk_")
                or isinstance(source, int)
            ):
```

```
                source_id = f"chunk_{source}"
            else:
                source_id = str(source)

            if (
                isinstance(target, str)
                and not target.startswith("chunk_")
                or isinstance(target, int)
            ):
                target_id = f"chunk_{target}"
            else:
                target_id = str(target)

            # Compute causal weight
            weight = self._compute_causal_weight(edge_type)

            if weight > 0:  # Only add edges with positive causal weight
                G.add_edge(
                    source_id,
                    target_id,
                    weight=weight,
                    edge_type=edge_type,
                    original_type=edge_type,
                )

        # Validate and clean graph
        if not nx.is_directed_acyclic_graph(G):
            logger.warning("Graph contains cycles, attempting to remove cycles")
            G = self._remove_cycles(G)

        logger.info(
            f"Built causal graph: {G.number_of_nodes()} nodes, "
            f"{G.number_of_edges()} edges, "
            f"is_dag={nx.is_directed_acyclic_graph(G)}"
        )

        return G

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight"
    )
    def _compute_causal_weight(self, edge_type: str) -> float:
        """
        Map SPC edge type to causal weight.

        Args:
            edge_type: Type of edge from SPC graph

        Returns:
            Causal weight between ParameterLoaderV2.get("farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight", "auto_param_L149_34", 0.0) and ParameterLoaderV2.get("farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight", "auto_param_L149_42", 1.0)
        """
        return self.CAUSAL_WEIGHTS.get(
            edge_type,
            ParameterLoaderV2.get(
                "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight",
```

```
                "auto_param_L151_50",
                0.0,
            ),
        )

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._remove_cycles"
    )
    def _remove_cycles(self, G: Any) -> Any:
        """
        Remove cycles from graph to create a DAG.

        Uses a simple strategy: remove edges with lowest weight until acyclic.

        Args:
            G: NetworkX DiGraph

        Returns:
            Modified graph (DAG)
        """
        if not HAS_NETWORKX:
            return G

        # Make a copy to avoid modifying original
        G_dag = G.copy()

        # Find cycles and remove lowest-weight edges
        while not nx.is_directed_acyclic_graph(G_dag):
            try:
                # Find a cycle
                cycle = nx.find_cycle(G_dag, orientation="original")

                # Find edge in cycle with minimum weight
                min_weight = float("inf")
                min_edge = None

                for u, v, direction in cycle:
                    if direction == "forward":
                        weight = G_dag[u][v].get(
                            "weight",
                            ParameterLoaderV2.get(
                                "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._remove_cycles",
                                "auto_param_L184_59",
                                0.0,
                            ),
                        )
                        if weight < min_weight:
                            min_weight = weight
                            min_edge = (u, v)

                # Remove the edge
                if min_edge:
                    logger.info(
                        f"Removing edge {min_edge} (weight={min_weight}) to break cycle"
                    )
                    G_dag.remove_edge(*min_edge)
```

```
                else:
                    # Shouldn’t happen, but break to avoid infinite loop
                    logger.error("Could not find edge to remove from cycle")
                    break

            except nx.NetworkXNoCycle:
                # No more cycles
                break

        return G_dag

    @calibrated_method(
        "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge.enhance_graph_with_content"
    )
    def enhance_graph_with_content(self, G: Any, chunks: list) -> Any:
        """
        Enhance causal graph with content-based relationships.

        This method can add additional edges based on content similarity,
        shared entities, or other semantic relationships.

        Args:
            G: NetworkX DiGraph (causal graph)
            chunks: List of ChunkData objects

        Returns:
            Enhanced graph
        """
        if not HAS_NETWORKX or G is None:
            return G

        _ = chunks  # Future enhancement: Add content-based edges using chunks
        return G



================================================================================
FILE: src/farfan_pipeline/analysis/teoria_cambio.py
================================================================================

#!/usr/bin/env python3
"""
Framework Unificado para la ValidaciÃ³n Causal de PolÃticas PÃºblicas
===================================================================

Este script consolida un conjunto de herramientas de nivel industrial en un
framework cohesivo, diseÃ±ado para la validaciÃ³n rigurosa de teorÃas de cambio
y modelos causales (DAGs). Su propÃ³sito es servir como el motor de anÃ¡lisis
estructural y estocÃ¡stico dentro de un flujo canÃ³nico de evaluaciÃ³n de planes
de desarrollo, garantizando que las polÃticas pÃºblicas no solo sean lÃ³gicamente
coherentes, sino tambiÃ©n estadÃsticamente robustas.

Arquitectura de Vanguardia:
---------------------------
1.  **Motor AxiomÃ¡tico de TeorÃa de Cambio (‘TeoriaCambio‘):**
    Valida la adherencia de un modelo a una jerarquÃa causal predefinida
```

```
    (Insumos â\206\222 Procesos â\206\222 Productos â\206\222 Resultados â\206\222 Causalidad), reflejando las
    dimensiones de evaluaciÃ³n (D1-D6) del flujo canÃ³nico.

2.  **Validador EstocÃ¡stico Avanzado (‘AdvancedDAGValidator‘):**
    Somete los modelos causales a un escrutinio probabilÃstico mediante
    simulaciones Monte Carlo deterministas. EvalÃºa la aciclicidad, la
    robustez estructural y el poder estadÃstico de la teorÃa.

3.  **Orquestador de CertificaciÃ³n Industrial (‘IndustrialGradeValidator‘):**
    Audita el rendimiento y la correctitud de la implementaciÃ³n del motor
    axiomÃ¡tico, asegurando que la herramienta de validaciÃ³n misma cumple con
    estÃ¡ndares de producciÃ³n.

4.  **Interfaz de LÃnea de Comandos (CLI):**
    Expone la funcionalidad a travÃ©s de una CLI robusta, permitiendo su
    integraciÃ³n en flujos de trabajo automatizados y su uso como herramienta
    de anÃ¡lisis configurable.

Autor: Sistema de ValidaciÃ³n de Planes de Desarrollo
VersiÃ³n: 4.0.0 (Refactorizada y Alineada)
Python: 3.10+
"""

# ============================================================================
# 1. IMPORTS Y CONFIGURACIÃ\223N GLOBAL
# ============================================================================

import argparse
import hashlib
import json
import logging
import random
import sys
import time
from collections import defaultdict, deque
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum, auto
from functools import lru_cache
from pathlib import Path
from typing import Any, ClassVar, Optional

# --- Dependencias de Terceros ---
import networkx as nx
import numpy as np
from scipy import stats

try:
    from jsonschema import Draft7Validator
except ImportError:  # pragma: no cover - jsonschema es opcional
    Draft7Validator = None

# CategoriaCausal moved to farfan_core.core.types to break architectural dependency
# (core.orchestrator was importing from analysis, which violates layer rules)
from farfan_pipeline.core.calibration.decorators import calibrated_method
from farfan_pipeline.core.parameters import ParameterLoaderV2
```

```
from farfan_pipeline.core.types import CategoriaCausal


# --- ConfiguraciÃ³n de Logging ---
def configure_logging() -> None:
    """Configura un sistema de logging de alto rendimiento para la salida estÃ¡ndar."""
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s.%(msecs)03d | %(levelname)-8s | %(name)s:%(lineno)d - %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
        stream=sys.stdout,
    )


configure_logging()
LOGGER = logging.getLogger(__name__)

# --- Constantes Globales ---
SEED: int = 42
STATUS_PASSED = "â\234\205 PASÃ\223"

# ============================================================================
# 2. ENUMS Y ESTRUCTURAS DE DATOS (DATACLASSES)
# ============================================================================


class GraphType(Enum):
    """TipologÃ-a de grafos para la aplicaciÃ³n de anÃ¡lisis especializados."""

    CAUSAL_DAG = auto()
    BAYESIAN_NETWORK = auto()
    STRUCTURAL_MODEL = auto()
    THEORY_OF_CHANGE = auto()


@dataclass
class ValidacionResultado:
    """Encapsula el resultado de la validaciÃ³n estructural de una teorÃ-a de cambio."""

    es_valida: bool = False
    violaciones_orden: list[tuple[str, str]] = field(default_factory=list)
    caminos_completos: list[list[str]] = field(default_factory=list)
    categorias_faltantes: list[CategoriaCausal] = field(default_factory=list)
    sugerencias: list[str] = field(default_factory=list)


@dataclass
class ValidationMetric:
    """Define una mÃ©trica de validaciÃ³n con umbrales y ponderaciÃ³n."""

    name: str
    value: float
    unit: str
    threshold: float
    status: str
    weight: float = 1.0
```

```
@dataclass
class AdvancedGraphNode:
    """Nodo de grafo enriquecido con metadatos y rol semÃ¡ntico."""

    name: str
    dependencies: set[str] = field(default_factory=set)
    metadata: dict[str, Any] = field(default_factory=dict)
    role: str = "variable"

    ALLOWED_ROLES: ClassVar[set[str]] = {
        "variable",
        "insumo",
        "proceso",
        "producto",
        "resultado",
        "causalidad",
    }

    def __post_init__(self) -> None:
        """Inicializa metadatos por defecto si no son provistos."""
        self.name = str(self.name).strip()
        if not self.name:
            raise ValueError("AdvancedGraphNode.name must be a non-empty string")

        if not isinstance(self.dependencies, set):
            self.dependencies = set(self.dependencies or set())
        self.dependencies = {
            str(dep).strip() for dep in self.dependencies if str(dep).strip()
        }

        self.metadata = self._normalize_metadata(self.metadata)

        normalized_role = (self.role or "variable").strip().lower()
        if normalized_role not in self.ALLOWED_ROLES:
            raise ValueError(
                "Invalid role '{}'. Expected one of: {}".format(
                    self.role, ", ".join(sorted(self.ALLOWED_ROLES))
                )
            )
        self.role = normalized_role

    def _normalize_metadata(
        self, metadata: dict[str, Any] | None = None
    ) -> dict[str, Any]:
        """Normaliza metadatos garantizando primitivos JSON y valores por defecto."""

        source_metadata = metadata if metadata is not None else self.metadata
        base_metadata = dict(source_metadata or {})
        if not base_metadata.get("created"):
            base_metadata["created"] = datetime.now().isoformat()
        if "confidence" not in base_metadata or base_metadata["confidence"] is None:
            base_metadata["confidence"] = ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
                "auto_param_L174_42",
```

```
                1.0,
            )

        normalized: dict[str, Any] = {}
        for key, value in base_metadata.items():
            if key == "confidence":
                normalized[key] = self._sanitize_confidence(value)
            elif key == "created":
                normalized[key] = self._sanitize_created(value)
            else:
                normalized[key] = self._sanitize_metadata_value(value)
        return normalized

    @staticmethod
    def _sanitize_confidence(value: Any) -> float:
        try:
            numeric = float(value)
        except (TypeError, ValueError):
            numeric = ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
                "numeric",
                1.0,
            )  # Refactored
        return max(
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
                "auto_param_L192_19",
                0.0,
            ),
            min(
                ParameterLoaderV2.get(
                    "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
                    "auto_param_L192_28",
                    1.0,
                ),
                numeric,
            ),
        )

    @staticmethod
    def _sanitize_created(value: Any) -> str:
        if isinstance(value, str) and value:
            return value
        if hasattr(value, "isoformat"):
            try:
                return value.isoformat()
            except Exception:  # pragma: no cover - fallback defensivo
                pass
        return datetime.now().isoformat()

    @staticmethod
    def _sanitize_metadata_value(value: Any) -> Any:
        if isinstance(value, (str, int, float, bool)) or value is None:
            return value
        if hasattr(value, "isoformat"):
            try:
```

```
                return value.isoformat()
            except Exception:  # pragma: no cover - fallback defensivo
                pass
        return str(value)

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.to_serializable_dict"
    )
    def to_serializable_dict(self) -> dict[str, Any]:
        """Convierte el nodo en un diccionario serializable compatible con JSON Schema."""

        metadata = self._normalize_metadata()
        return {
            "name": self.name,
            "dependencies": sorted(self.dependencies),
            "metadata": metadata,
            "role": self.role,
        }


@dataclass
class MonteCarloAdvancedResult:
    """
    Resultado exhaustivo de una simulaciÃ³n Monte Carlo.

    Audit Point 1.1: Deterministic Seeding (RNG)
    Field 'reproducible' confirms that seed was deterministically generated
    and results can be reproduced with identical inputs.
    """

    plan_name: str
    seed: int  # Audit 1.1: Deterministic seed from _create_advanced_seed
    timestamp: str
    total_iterations: int
    acyclic_count: int
    p_value: float
    bayesian_posterior: float
    confidence_interval: tuple[float, float]
    statistical_power: float
    edge_sensitivity: dict[str, float]
    node_importance: dict[str, float]
    robustness_score: float
    reproducible: bool  # Audit 1.1: True when deterministic seed used
    convergence_achieved: bool
    adequate_power: bool
    computation_time: float
    graph_statistics: dict[str, Any]
    test_parameters: dict[str, Any]


# ============================================================================
# 3. MOTOR AXIOMÃ\201TICO DE TEORÃ\215A DE CAMBIO
# ============================================================================


class TeoriaCambio:
```

```
    """
    Motor para la construcciÃ³n y validaciÃ³n estructural de teorÃas de cambio.
    Valida la coherencia lÃ³gica de grafos causales contra un modelo axiomÃ¡tico
    de categorÃas jerÃ¡rquicas, crucial para el anÃ¡lisis de polÃticas pÃºblicas.
    """

    _MATRIZ_VALIDACION: dict[CategoriaCausal, frozenset[CategoriaCausal]] = {
        cat: (
            frozenset({cat, CategoriaCausal(cat.value + 1)})
            if cat.value < 5
            else frozenset({cat})
        )
        for cat in CategoriaCausal
    }

    def __init__(self) -> None:
        """Inicializa el motor con un sistema de cache optimizado."""
        self._grafo_cache: nx.DiGraph | None = None
        self._cache_valido: bool = False
        self.logger: logging.Logger = LOGGER

    @staticmethod
    def _es_conexion_valida(origen: CategoriaCausal, destino: CategoriaCausal) -> bool:
        """Verifica la validez de una conexiÃ³n causal segÃºn la jerarquÃa estructural."""
        return destino in TeoriaCambio._MATRIZ_VALIDACION.get(origen, frozenset())

    @lru_cache(maxsize=128)
    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.TeoriaCambio.construir_grafo_causal"
    )
    def construir_grafo_causal(self) -> nx.DiGraph:
        """Construye y cachea el grafo causal canÃ³nico."""
        if self._grafo_cache is not None and self._cache_valido:
            self.logger.debug("Recuperando grafo causal desde cachÃ©.")
            return self._grafo_cache

        grafo = nx.DiGraph()
        for cat in CategoriaCausal:
            grafo.add_node(cat.name, categoria=cat, nivel=cat.value)
        for origen in CategoriaCausal:
            for destino in self._MATRIZ_VALIDACION.get(origen, frozenset()):
                if origen != destino:
                    grafo.add_edge(
                        origen.name,
                        destino.name,
                        peso=ParameterLoaderV2.get(
                            "farfan_core.analysis.teoria_cambio.TeoriaCambio.construir_grafo_causal",
                            "auto_param_L302_67",
                            1.0,
                        ),
                    )

        self._grafo_cache = grafo
        self._cache_valido = True
        self.logger.info(
            "Grafo causal canÃ³nico construido: %d nodos, %d aristas.",
```

```
            grafo.number_of_nodes(),
            grafo.number_of_edges(),
        )
        return grafo

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.TeoriaCambio.construir_grafo_from_cpp"
    )
    def construir_grafo_from_cpp(self, cpp) -> nx.DiGraph:
        """
        Construir grafo causal desde CanonPolicyPackage (Phase 1 output).

        Este mÃ©todo integra el Phase 1-2 adapter, permitiendo construir grafos
        causales directamente desde el CanonPolicyPackage para anÃ¡lisis de
        TeorÃ-a de Cambio en dimensiones D4 y D6.

        Args:
            cpp: CanonPolicyPackage from Phase 1 ingestion

        Returns:
            NetworkX DiGraph con relaciones causales derivadas de chunk_graph

        Raises:
            ValueError: If cpp is invalid
        """
        try:
            from farfan_pipeline.analysis.spc_causal_bridge import SPCCausalBridge

            bridge = SPCCausalBridge()
            causal_graph = bridge.build_causal_graph_from_cpp(cpp)

            if causal_graph is None:
                self.logger.warning(
                    "Failed to build causal graph from CPP, using standard graph"
                )
                return self.construir_grafo_causal()

            self.logger.info(
                "Grafo causal construido desde CPP: %d nodos, %d aristas.",
                causal_graph.number_of_nodes(),
                causal_graph.number_of_edges(),
            )

            return causal_graph

        except (ImportError, ValueError) as e:
            self.logger.error(f"Error building causal graph from CPP: {e}")
            return self.construir_grafo_causal()

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.TeoriaCambio.construir_grafo_from_spc"
    )
    def construir_grafo_from_spc(self, preprocessed_doc) -> nx.DiGraph:
        """
        Construir grafo causal desde estructura SPC (Smart Policy Chunks).

```

```
        Este mÃ©todo permite construir grafos causales a partir de la estructura
        semÃ¡ntica preservada por SPC, en lugar de extraer relaciones causales
        Ãºnicamente del texto.

        Args:
            preprocessed_doc: PreprocessedDocument con modo chunked

        Returns:
            NetworkX DiGraph con relaciones causales derivadas de SPC
        """
        # Check if document is in chunked mode
        if getattr(preprocessed_doc, "processing_mode", "flat") != "chunked":
            # Fallback to text-based construction for flat mode
            self.logger.warning(
                "Document not in chunked mode, using standard causal graph"
            )
            return self.construir_grafo_causal()

        try:
            from farfan_pipeline.analysis.spc_causal_bridge import SPCCausalBridge

            # Use SPC bridge to construct base graph
            bridge = SPCCausalBridge()
            chunk_graph = getattr(preprocessed_doc, "chunk_graph", {})

            if not chunk_graph:
                self.logger.warning(
                    "No chunk graph available, using standard causal graph"
                )
                return self.construir_grafo_causal()

            base_graph = bridge.build_causal_graph_from_spc(chunk_graph)

            if base_graph is None:
                self.logger.warning(
                    "Failed to build SPC graph, using standard causal graph"
                )
                return self.construir_grafo_causal()

            # Enhance with content analysis from chunks
            chunks = getattr(preprocessed_doc, "chunks", [])
            if chunks:
                base_graph = bridge.enhance_graph_with_content(base_graph, chunks)

            self.logger.info(
                "Grafo causal SPC construido: %d nodos, %d aristas.",
                base_graph.number_of_nodes(),
                base_graph.number_of_edges(),
            )

            return base_graph

        except ImportError as e:
            self.logger.error(f"SPCCausalBridge not available: {e}")
            return self.construir_grafo_causal()

```

```
    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.TeoriaCambio.validacion_completa"
    )
    def validacion_completa(self, grafo: nx.DiGraph) -> ValidacionResultado:
        """Ejecuta una validaciÃ³n estructural exhaustiva de la teorÃa de cambio."""
        resultado = ValidacionResultado()
        categorias_presentes = self._extraer_categorias(grafo)
        resultado.categorias_faltantes = [
            c for c in CategoriaCausal if c.name not in categorias_presentes
        ]
        resultado.violaciones_orden = self._validar_orden_causal(grafo)
        resultado.caminos_completos = self._encontrar_caminos_completos(grafo)
        resultado.es_valida = not (
            resultado.categorias_faltantes or resultado.violaciones_orden
        ) and bool(resultado.caminos_completos)
        resultado.sugerencias = self._generar_sugerencias_internas(resultado)
        return resultado

    @staticmethod
    def _extraer_categorias(grafo: nx.DiGraph) -> set[str]:
        """Extrae el conjunto de categorÃas presentes en el grafo."""
        return {
            data["categoria"].name
            for _, data in grafo.nodes(data=True)
            if "categoria" in data
        }

    @staticmethod
    def _validar_orden_causal(grafo: nx.DiGraph) -> list[tuple[str, str]]:
        """Identifica las aristas que violan el orden causal axiomÃ¡tico."""
        violaciones = []
        for u, v in grafo.edges():
            cat_u = grafo.nodes[u].get("categoria")
            cat_v = grafo.nodes[v].get("categoria")
            if cat_u and cat_v and not TeoriaCambio._es_conexion_valida(cat_u, cat_v):
                violaciones.append((u, v))
        return violaciones

    @staticmethod
    def _encontrar_caminos_completos(grafo: nx.DiGraph) -> list[list[str]]:
        """Encuentra todos los caminos simples desde nodos INSUMOS a CAUSALIDAD."""
        try:
            nodos_inicio = [
                n
                for n, d in grafo.nodes(data=True)
                if d.get("categoria") == CategoriaCausal.INSUMOS
            ]
            nodos_fin = [
                n
                for n, d in grafo.nodes(data=True)
                if d.get("categoria") == CategoriaCausal.CAUSALIDAD
            ]
            return [
                path
                for u in nodos_inicio
                for v in nodos_fin
```

```
                for path in nx.all_simple_paths(grafo, u, v)
            ]
        except Exception as e:
            LOGGER.warning("Fallo en la detecciÃ³n de caminos completos: %s", e)
            return []

    @staticmethod
    def _generar_sugerencias_internas(validacion: ValidacionResultado) -> list[str]:
        """Genera un listado de sugerencias accionables basadas en los resultados."""
        sugerencias = []
        if validacion.categorias_faltantes:
            sugerencias.append(
                f"Integridad estructural comprometida. Incorporar: {’, ’.join(c.name for c in validacion.categorias_faltantes)}."
            )
        if validacion.violaciones_orden:
            sugerencias.append(
                f"Corregir {len(validacion.violaciones_orden)} violaciones de secuencia causal para restaurar la coherencia lÃ³gica."
            )
        if not validacion.caminos_completos:
            sugerencias.append(
                "La teorÃa es incompleta. Establecer al menos un camino causal de INSUMOS a CAUSALIDAD."
            )
        if validacion.es_valida:
            sugerencias.append(
                "La teorÃa es estructuralmente vÃ¡lida. Proceder con anÃ¡lisis de robustez estocÃ¡stica."
            )
        return sugerencias

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.TeoriaCambio._execute_generar_sugerencias_internas"
    )
    def _execute_generar_sugerencias_internas(
        self, validacion: "ValidacionResultado"
    ) -> list[str]:
        """
        Execute internal suggestion generation (wrapper method).

        This method wraps the static _generar_sugerencias_internas method
        to allow it to be called via the method executor interface.

        Args:
            validacion: Validation result object

        Returns:
            List of actionable suggestions
        """
        return self._generar_sugerencias_internas(validacion)


# ============================================================================
# 4. VALIDADOR ESTOCÃ\201STICO AVANZADO DE DAGs
# ============================================================================


def _create_advanced_seed(plan_name: str, salt: str = "") -> int:
    """
```

```
    Genera una semilla determinista de alta entropÃa usando SHA-512.

    Audit Point 1.1: Deterministic Seeding (RNG)
    Global random seed generated deterministically from plan_name and optional salt.
    Confirms reproducibility across numpy/torch/PyMC stochastic elements.

    Args:
        plan_name: Plan identifier for deterministic derivation
        salt: Optional salt for sensitivity analysis (varies to bound variance)

    Returns:
        64-bit unsigned integer seed derived from SHA-512 hash

    Quality Evidence:
        Re-run pipeline twice with identical inputs/salt â\206\222 output hashes must match 100%
        Achieves MMR-level determinism per Beach & Pedersen 2019
    """
    combined = f"{plan_name}-{salt}".encode()
    hash_obj = hashlib.sha512(combined)
    seed = int.from_bytes(hash_obj.digest()[:8], "big", signed=False)

    # Log for audit trail
    LOGGER.info(
        f"[Audit 1.1] Deterministic seed: {seed} (plan={plan_name}, salt={salt})"
    )

    return seed


class AdvancedDAGValidator:
    """
    Motor para la validaciÃ³n estocÃ¡stica y anÃ¡lisis de sensibilidad de DAGs.
    Utiliza simulaciones Monte Carlo para cuantificar la robustez y aciclicidad
    de modelos causales complejos.
    """

    _NODE_SCHEMA_PATH: Path = (
        Path(__file__).resolve().parent
        / "schemas"
        / "teoria_cambio"
        / "advanced_graph_node.schema.json"
    )
    _NODE_VALIDATOR: Any | None = None
    _NODE_VALIDATION_WARNING_EMITTED: bool = False

    def __init__(self, graph_type: GraphType = GraphType.CAUSAL_DAG) -> None:
        self.graph_nodes: dict[str, AdvancedGraphNode] = {}
        self.graph_type: GraphType = graph_type
        self._rng: random.Random | None = None
        self.config: dict[str, Any] = {
            "default_iterations": 10000,
            "confidence_level": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.__init__",
                "auto_param_L517_32",
                0.95,
            ),
```

```
            "power_threshold": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.__init__",
                "auto_param_L518_31",
                0.8,
            ),
            "convergence_threshold": 1e-5,
        }
        self._last_serialized_nodes: list[dict[str, Any]] = []

    def add_node(
        self,
        name: str,
        dependencies: set[str] | None = None,
        role: str = "variable",
        metadata: dict[str, Any] | None = None,
    ) -> None:
        """Agrega un nodo enriquecido al grafo."""
        self.graph_nodes[name] = AdvancedGraphNode(
            name, dependencies or set(), metadata or {}, role
        )

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.add_edge"
    )
    def add_edge(self, from_node: str, to_node: str, weight: float = 1.0) -> None:
        """Agrega una arista dirigida con peso opcional."""
        if to_node not in self.graph_nodes:
            self.add_node(to_node)
        if from_node not in self.graph_nodes:
            self.add_node(from_node)
        self.graph_nodes[to_node].dependencies.add(from_node)
        self.graph_nodes[to_node].metadata[f"edge_{from_node}->{to_node}"] = weight

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._initialize_rng"
    )
    def _initialize_rng(self, plan_name: str, salt: str = "") -> int:
        """
        Inicializa el generador de nÃºmeros aleatorios con una semilla determinista.

        Audit Point 1.1: Deterministic Seeding (RNG)
        Initializes numpy/random RNG with deterministic seed for reproducibility.
        Sets reproducible=True in MonteCarloAdvancedResult.

        Args:
            plan_name: Plan identifier for seed derivation
            salt: Optional salt for sensitivity analysis

        Returns:
            Generated seed value for audit logging
        """
        seed = _create_advanced_seed(plan_name, salt)
        self._rng = random.Random(seed)
        np.random.seed(seed % (2**32))

        # Log initialization for reproducibility verification
```

```
        LOGGER.info(
            f"[Audit 1.1] RNG initialized with seed={seed} for plan={plan_name}"
        )

        return seed

    @staticmethod
    def _is_acyclic(nodes: dict[str, AdvancedGraphNode]) -> bool:
        """DetecciÃ³n de ciclos mediante el algoritmo de Kahn (ordenaciÃ³n topolÃ³gica)."""
        if not nodes:
            return True
        in_degree = dict.fromkeys(nodes, 0)
        adjacency = defaultdict(list)
        for name, node in nodes.items():
            for dep in node.dependencies:
                if dep in nodes:
                    adjacency[dep].append(name)
                    in_degree[name] += 1

        queue = deque([name for name, degree in in_degree.items() if degree == 0])
        count = 0
        while queue:
            u = queue.popleft()
            count += 1
            for v in adjacency[u]:
                in_degree[v] -= 1
                if in_degree[v] == 0:
                    queue.append(v)
        return count == len(nodes)

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._generate_subgraph"
    )
    def _generate_subgraph(self) -> dict[str, AdvancedGraphNode]:
        """Genera un subgrafo aleatorio del grafo principal."""
        if not self.graph_nodes or self._rng is None:
            return {}
        node_count = len(self.graph_nodes)
        subgraph_size = self._rng.randint(min(3, node_count), node_count)
        selected_names = self._rng.sample(list(self.graph_nodes.keys()), subgraph_size)

        subgraph = {}
        selected_set = set(selected_names)
        for name in selected_names:
            original = self.graph_nodes[name]
            subgraph[name] = AdvancedGraphNode(
                name,
                original.dependencies.intersection(selected_set),
                original.metadata.copy(),
                original.role,
            )
        return subgraph

    def calculate_acyclicity_pvalue(
        self, plan_name: str, iterations: int
    ) -> MonteCarloAdvancedResult:
```

```
        """CÃ¡lculo avanzado de p-value con un marco estadÃ-stico completo."""
        start_time = time.time()
        seed = self._initialize_rng(plan_name)
        if not self.graph_nodes:
            self._last_serialized_nodes = []
            return self._create_empty_result(
                plan_name, seed, datetime.now().isoformat()
            )

        acyclic_count = sum(
            1 for _ in range(iterations) if self._is_acyclic(self._generate_subgraph())
        )

        p_value = (
            acyclic_count / iterations
            if iterations > 0
            else ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._generate_subgraph",
                "auto_param_L633_68",
                1.0,
            )
        )
        conf_level = self.config["confidence_level"]
        ci = self._calculate_confidence_interval(acyclic_count, iterations, conf_level)
        power = self._calculate_statistical_power(acyclic_count, iterations)

        # AnÃ¡lisis de Sensibilidad (simplificado para el flujo principal)
        sensitivity = self._perform_sensitivity_analysis_internal(
            plan_name, p_value, min(iterations, 200)
        )

        self.export_nodes(validate=True)

        return MonteCarloAdvancedResult(
            plan_name=plan_name,
            seed=seed,
            timestamp=datetime.now().isoformat(),
            total_iterations=iterations,
            acyclic_count=acyclic_count,
            p_value=p_value,
            bayesian_posterior=self._calculate_bayesian_posterior(p_value),
            confidence_interval=ci,
            statistical_power=power,
            edge_sensitivity=sensitivity.get("edge_sensitivity", {}),
            node_importance=self._calculate_node_importance(),
            robustness_score=1 / (1 + sensitivity.get("average_sensitivity", 0)),
            reproducible=True,  # La reproducibilidad es por diseÃ±o de la semilla
            convergence_achieved=(p_value * (1 - p_value) / iterations)
              < self.config["convergence_threshold"],
            adequate_power=power >= self.config["power_threshold"],
            computation_time=time.time() - start_time,
            graph_statistics=self.get_graph_stats(),
            test_parameters={"iterations": iterations, "confidence_level": conf_level},
        )

    @property
```

```
    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes"
    )
    def last_serialized_nodes(self) -> list[dict[str, Any]]:
        """Obtiene la instantÃ¡nea mÃ¡s reciente de nodos serializados."""

        return [
            {
                "name": node["name"],
                "dependencies": list(node["dependencies"]),
                "metadata": dict(node["metadata"]),
                "role": node["role"],
            }
            for node in self._last_serialized_nodes
        ]

    def export_nodes(
        self, validate: bool = False, schema_path: Path | None = None
    ) -> list[dict[str, Any]]:
        """Serializa los nodos del grafo y opcionalmente valida contra JSON Schema."""

        serialized_nodes = [
            node.to_serializable_dict()
            for node in sorted(self.graph_nodes.values(), key=lambda n: n.name)
        ]
        self._last_serialized_nodes = serialized_nodes

        if validate:
            validator = self._get_node_validator(schema_path)
            if validator is not None:
                for index, payload in enumerate(serialized_nodes):
                    errors = list(validator.iter_errors(payload))
                    if errors:
                        joined = "; ".join(
                            (
                                f"{’/’.join(str(x) for x in error.path)}: {error.message}"
                                if error.path
                                else error.message
                            )
                            for error in errors
                        )
                        raise ValueError(
                            "AdvancedGraphNode payload at index %d failed schema validation: %s"
                            % (index, joined)
                        )

        return serialized_nodes

    @classmethod
    def _get_node_validator(
        cls, schema_path: Path | None = None
    ) -> Optional["Draft7Validator"]:
        """Obtiene (y cachea) el validador JSON Schema para nodos avanzados."""

        if Draft7Validator is None:
            if not cls._NODE_VALIDATION_WARNING_EMITTED:
```

```
                LOGGER.warning(
                    "jsonschema is not installed; skipping AdvancedGraphNode schema validation."
                )
                cls._NODE_VALIDATION_WARNING_EMITTED = True
            return None

        if schema_path is None and cls._NODE_VALIDATOR is not None:
            return cls._NODE_VALIDATOR

        path = Path(schema_path) if schema_path else cls._NODE_SCHEMA_PATH

        # Delegate to factory for I/O operation
        from farfan_pipeline.analysis.factory import load_json

        try:
            schema = load_json(path)
        except FileNotFoundError:
            LOGGER.error("Advanced graph node schema file not found at %s", path)
            return None
        except json.JSONDecodeError as exc:
            LOGGER.error("Invalid JSON in advanced graph node schema %s: %s", path, exc)
            return None

        validator = Draft7Validator(schema)
        if schema_path is None:
            cls._NODE_VALIDATOR = validator
        return validator

    def _perform_sensitivity_analysis_internal(
        self, plan_name: str, base_p_value: float, iterations: int
    ) -> dict[str, Any]:
        """AnÃ¡lisis de sensibilidad interno optimizado para evitar cÃ¡lculos redundantes."""
        edge_sensitivity: dict[str, float] = {}
        # 1. Genera los subgrafos una sola vez
        subgraphs = []
        for _ in range(iterations):
            subgraph = self._generate_subgraph()
            subgraphs.append(subgraph)
        # 2. Lista de todas las aristas
        edges = {
            f"{dep}->{name}"
            for name, node in self.graph_nodes.items()
            for dep in node.dependencies
        }
        # 3. Para cada arista, calcula el p-value perturbado usando los mismos subgrafos
        for edge in edges:
            from_node, to_node = edge.split("->")
            acyclic_count = 0
            for subgraph in subgraphs:
                # Perturba el subgrafo removiendo la arista
                if to_node in subgraph and from_node in subgraph[to_node].dependencies:
                    subgraph_copy = {
                        k: AdvancedGraphNode(
                            v.name, set(v.dependencies), dict(v.metadata), v.role
                        )
                        for k, v in subgraph.items()
```

```
                    }
                    subgraph_copy[to_node].dependencies.discard(from_node)
                else:
                    subgraph_copy = subgraph
                if AdvancedDAGValidator._is_acyclic(subgraph_copy):
                    acyclic_count += 1
            perturbed_p = acyclic_count / iterations
            edge_sensitivity[edge] = abs(base_p_value - perturbed_p)
        sens_values = list(edge_sensitivity.values())
        return {
            "edge_sensitivity": edge_sensitivity,
            "average_sensitivity": np.mean(sens_values) if sens_values else 0,
        }

    @staticmethod
    def _calculate_confidence_interval(
        s: int, n: int, conf: float
    ) -> tuple[float, float]:
        """Calcula el intervalo de confianza de Wilson."""
        if n == 0:
            return (
                ParameterLoaderV2.get(
                    "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
                    "auto_param_L793_20",
                    0.0,
                ),
                ParameterLoaderV2.get(
                    "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
                    "auto_param_L793_25",
                    1.0,
                ),
            )
        z = stats.norm.ppf(1 - (1 - conf) / 2)
        p_hat = s / n
        den = 1 + z**2 / n
        center = (p_hat + z**2 / (2 * n)) / den
        width = (z * np.sqrt(p_hat * (1 - p_hat) / n + z**2 / (4 * n**2))) / den
        return (max(0, center - width), min(1, center + width))

    @staticmethod
    def _calculate_statistical_power(
        s: int,
        n: int,
        alpha: float = ParameterLoaderV2.get(
            "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
            "auto_param_L802_68",
            0.05,
        ),
    ) -> float:
        """Calcula el poder estadÃstico a posteriori."""
        if n == 0:
            return ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
                "auto_param_L805_19",
                0.0,
            )
```

```
        p = s / n
        effect_size = 2 * (
            np.arcsin(np.sqrt(p))
            - np.arcsin(
                np.sqrt(
                    ParameterLoaderV2.get(
                        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
                        "auto_param_L807_69",
                        0.5,
                    )
                )
            )
        )
        return stats.norm.sf(
            stats.norm.ppf(1 - alpha) - abs(effect_size) * np.sqrt(n / 2)
        )

    @staticmethod
    def _calculate_bayesian_posterior(
        likelihood: float,
        prior: float = ParameterLoaderV2.get(
            "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
            "auto_param_L813_72",
            0.5,
        ),
    ) -> float:
        """Calcula la probabilidad posterior Bayesiana simple."""
        if (likelihood * prior + (1 - likelihood) * (1 - prior)) == 0:
            return prior
        return (likelihood * prior) / (
            likelihood * prior + (1 - likelihood) * (1 - prior)
        )

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._calculate_node_importance"
    )
    def _calculate_node_importance(self) -> dict[str, float]:
        """Calcula una mÃ©trica de importancia para cada nodo."""
        if not self.graph_nodes:
            return {}
        out_degree = defaultdict(int)
        for node in self.graph_nodes.values():
            for dep in node.dependencies:
                out_degree[dep] += 1

        max_centrality = (
            max(
                len(node.dependencies) + out_degree[name]
                for name, node in self.graph_nodes.items()
            )
            or 1
        )
        return {
            name: (len(node.dependencies) + out_degree[name]) / max_centrality
            for name, node in self.graph_nodes.items()
        }
```

```

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats"
    )
    def get_graph_stats(self) -> dict[str, Any]:
        """Obtiene estadÃ-sticas estructurales del grafo."""
        nodes = len(self.graph_nodes)
        edges = sum(len(n.dependencies) for n in self.graph_nodes.values())
        return {
            "nodes": nodes,
            "edges": edges,
            "density": edges / (nodes * (nodes - 1)) if nodes > 1 else 0,
        }

    def _create_empty_result(
        self, plan_name: str, seed: int, timestamp: str
    ) -> MonteCarloAdvancedResult:
        """Crea un resultado vacÃ-o para grafos sin nodos."""
        return MonteCarloAdvancedResult(
            plan_name,
            seed,
            timestamp,
            0,
            0,
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                "auto_param_L864_12",
                1.0,
            ),
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                "auto_param_L865_12",
                1.0,
            ),
            (
                ParameterLoaderV2.get(
                    "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                    "auto_param_L866_13",
                    0.0,
                ),
                ParameterLoaderV2.get(
                    "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                    "auto_param_L866_18",
                    1.0,
                ),
            ),
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                "auto_param_L867_12",
                0.0,
            ),
            {},
            {},
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                "auto_param_L870_12",
```

```
                1.0,
            ),
            True,
            True,
            False,
            ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
                "auto_param_L874_12",
                0.0,
            ),
            {},
            {},
        )


# ============================================================================
# 5. ORQUESTADOR DE CERTIFICACIÃ\223N INDUSTRIAL
# ============================================================================


class IndustrialGradeValidator:
    """
    Orquesta una validaciÃ³n de grado industrial para el motor de TeorÃa de Cambio.
    """

    def __init__(self) -> None:
        self.logger: logging.Logger = LOGGER
        self.metrics: list[ValidationMetric] = []
        self.performance_benchmarks: dict[str, float] = {
            "engine_readiness": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
                "auto_param_L892_32",
                0.05,
            ),
            "graph_construction": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
                "auto_param_L893_34",
                0.1,
            ),
            "path_detection": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
                "auto_param_L894_30",
                0.2,
            ),
            "full_validation": ParameterLoaderV2.get(
                "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
                "auto_param_L895_31",
                0.3,
            ),
        }

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.execute_suite"
    )
    def execute_suite(self) -> bool:
        """Ejecuta la suite completa de validaciÃ³n industrial."""
```

```
        self.logger.info("=" * 80)
        self.logger.info("INICIO DE SUITE DE CERTIFICACIÃ\223N INDUSTRIAL")
        self.logger.info("=" * 80)
        start_time = time.time()

        results = [
            self.validate_engine_readiness(),
            self.validate_causal_categories(),
            self.validate_connection_matrix(),
            self.run_performance_benchmarks(),
        ]

        total_time = time.time() - start_time
        passed = sum(1 for m in self.metrics if m.status == STATUS_PASSED)
        success_rate = (passed / len(self.metrics) * 100) if self.metrics else 100

        self.logger.info("\n" + "=" * 80)
        self.logger.info("ð\237\223\212 INFORME DE CERTIFICACIÃ\223N INDUSTRIAL")
        self.logger.info("=" * 80)
        self.logger.info(f"  - Tiempo Total de la Suite: {total_time:.3f} segundos")
        self.logger.info(
            f"  - Tasa de Ã\211xito de MÃ©tricas: {success_rate:.1f}%% ({passed}/{len(self.metrics)})"
        )

        meets_standards = all(results) and success_rate >= 90.0
        self.logger.info(
            f"  ð\237\217\206 VEREDICTO: {'CERTIFICACIÃ\223N OTORGADA' if meets_standards else 'SE REQUIEREN MEJORAS'}"
        )
        return meets_standards

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_engine_readiness"
    )
    def validate_engine_readiness(self) -> bool:
        """Valida la disponibilidad y tiempo de instanciaciÃ³n de los motores de anÃ¡lisis."""
        self.logger.info("  [Capa 1] Validando disponibilidad de motores...")
        start_time = time.time()
        try:
            _ = TeoriaCambio()
            _ = AdvancedDAGValidator()
            instantiation_time = time.time() - start_time
            metric = self._log_metric(
                "Disponibilidad del Motor",
                instantiation_time,
                "s",
                self.performance_benchmarks["engine_readiness"],
            )
            return metric.status == STATUS_PASSED
        except Exception as e:
            self.logger.error("    â\235\214 Error crÃtico al instanciar motores: %s", e)
            return False

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_causal_categories"
    )
    def validate_causal_categories(self) -> bool:
```

```
        """Valida la completitud y el orden axiomÃ¡tico de las categorÃas causales."""
        self.logger.info("  [Capa 2] Validando axiomas de categorÃas causales...")
        expected = {cat.name: cat.value for cat in CategoriaCausal}
        if len(expected) != 5 or any(
            expected[name] != i + 1
            for i, name in enumerate(
                ["INSUMOS", "PROCESOS", "PRODUCTOS", "RESULTADOS", "CAUSALIDAD"]
            )
        ):
            self.logger.error(
                "    â\235\214 DefiniciÃ³n de CategoriaCausal es inconsistente con el axioma."
            )
            return False
        self.logger.info("    â\234\205 Axiomas de categorÃas validados.")
        return True

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_connection_matrix"
    )
    def validate_connection_matrix(self) -> bool:
        """Valida la matriz de transiciones causales."""
        self.logger.info("  [Capa 3] Validando matriz de transiciones causales...")
        tc = TeoriaCambio()
        errors = 0
        for o in CategoriaCausal:
            for d in CategoriaCausal:
                is_valid = tc._es_conexion_valida(o, d)
                expected = d in tc._MATRIZ_VALIDACION.get(o, set())
                if is_valid != expected:
                    errors += 1
        if errors > 0:
            self.logger.error(
                "    â\235\214 %d inconsistencias encontradas en la matriz de validaciÃ³n.",
                errors,
            )
            return False
        self.logger.info("    â\234\205 Matriz de transiciones validada.")
        return True

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.run_performance_benchmarks"
    )
    def run_performance_benchmarks(self) -> bool:
        """Ejecuta benchmarks de rendimiento para las operaciones crÃticas del motor."""
        self.logger.info("  [Capa 4] Ejecutando benchmarks de rendimiento...")
        tc = TeoriaCambio()

        grafo = self._benchmark_operation(
            "ConstrucciÃ³n de Grafo",
            tc.construir_grafo_causal,
            self.performance_benchmarks["graph_construction"],
        )
        _ = self._benchmark_operation(
            "DetecciÃ³n de Caminos",
            tc._encontrar_caminos_completos,
            self.performance_benchmarks["path_detection"],
```

```
            grafo,
        )
        _ = self._benchmark_operation(
            "ValidaciÃ³n Completa",
            tc.validacion_completa,
            self.performance_benchmarks["full_validation"],
            grafo,
        )

        return all(
            m.status == STATUS_PASSED
            for m in self.metrics
            if m.name in self.performance_benchmarks
        )

    def _benchmark_operation(
        self, operation_name: str, callable_obj, threshold: float, *args, **kwargs
    ):
        """Mide el tiempo de ejecuciÃ³n de una operaciÃ³n y registra la mÃ©trica."""
        start_time = time.time()
        result = callable_obj(*args, **kwargs)
        elapsed = time.time() - start_time
        self._log_metric(operation_name, elapsed, "s", threshold)
        return result

    @calibrated_method(
        "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator._log_metric"
    )
    def _log_metric(self, name: str, value: float, unit: str, threshold: float):
        """Registra y reporta una mÃ©trica de validaciÃ³n."""
        status = STATUS_PASSED if value <= threshold else "â\235\214 FALLÃ\223"
        metric = ValidationMetric(name, value, unit, threshold, status)
        self.metrics.append(metric)
        icon = "ð\237\237¢" if status == STATUS_PASSED else "ð\237\224´"
        self.logger.info(
            f"    {icon} {name}: {value:.4f} {unit} (LÃmite: {threshold:.4f} {unit}) - {status}"
        )
        return metric


# ============================================================================
# 6. LÃ\223GICA DE LA CLI Y CONSTRUCTORES DE GRAFOS DE DEMOSTRACIÃ\223N
# ============================================================================


def create_policy_theory_of_change_graph() -> AdvancedDAGValidator:
    """
    Construye un grafo causal de demostraciÃ³n alineado con la polÃtica P1:
    "Derechos de las mujeres e igualdad de gÃ©nero".
    """
    validator = AdvancedDAGValidator(graph_type=GraphType.THEORY_OF_CHANGE)

    # Nodos basados en el lexicÃ³n y las dimensiones D1-D5
    validator.add_node("recursos_financieros", role="insumo")
    validator.add_node(
        "mecanismos_de_adelanto", dependencies={"recursos_financieros"}, role="proceso"
```

```
    )
    validator.add_node(
        "comisarias_funcionales",
        dependencies={"mecanismos_de_adelanto"},
        role="producto",
    )
    validator.add_node(
        "reduccion_vbg", dependencies={"comisarias_funcionales"}, role="resultado"
    )
    validator.add_node(
        "aumento_participacion_politica",
        dependencies={"mecanismos_de_adelanto"},
        role="resultado",
    )
    validator.add_node(
        "autonomia_economica",
        dependencies={"reduccion_vbg", "aumento_participacion_politica"},
        role="causalidad",
    )

    LOGGER.info("Grafo de demostraciÃ³n para la polÃtica ’P1’ construido.")
    return validator


def main() -> None:
    """Punto de entrada principal para la interfaz de lÃnea de comandos (CLI)."""
    parser = argparse.ArgumentParser(
        description="Framework Unificado para la ValidaciÃ³n Causal de PolÃticas PÃºblicas.",
        formatter_class=argparse.RawTextHelpFormatter,
    )
    subparsers = parser.add_subparsers(dest="command", required=True)

    # --- Comando: industrial-check ---
    subparsers.add_parser(
        "industrial-check",
        help="Ejecuta la suite de certificaciÃ³n industrial sobre los motores de validaciÃ³n.",
    )

    # --- Comando: stochastic-validation ---
    parser_stochastic = subparsers.add_parser(
        "stochastic-validation",
        help="Ejecuta la validaciÃ³n estocÃ¡stica sobre un modelo causal de polÃtica.",
    )
    parser_stochastic.add_argument(
        "plan_name",
        type=str,
        help="Nombre del plan o polÃtica a validar (usado como semilla).",
    )
    parser_stochastic.add_argument(
        "-i",
        "--iterations",
        type=int,
        default=10000,
        help="NÃºmero de iteraciones para la simulaciÃ³n Monte Carlo.",
    )

```

```
    args = parser.parse_args()

    if args.command == "industrial-check":
        validator = IndustrialGradeValidator()
        success = validator.execute_suite()
        sys.exit(0 if success else 1)

    elif args.command == "stochastic-validation":
        LOGGER.info("Iniciando validaciÃ³n estocÃ¡stica para el plan: %s", args.plan_name)
        # Se podrÃa cargar un grafo desde un archivo, pero para la demo usamos el constructor
        dag_validator = create_policy_theory_of_change_graph()
        result = dag_validator.calculate_acyclicity_pvalue(
            args.plan_name, args.iterations
        )
        serialized_nodes = dag_validator.last_serialized_nodes

        LOGGER.info("\n" + "=" * 80)
        LOGGER.info(
            f"RESULTADOS DE LA VALIDACIÃ\223N ESTOCÃ\201STICA PARA '{result.plan_name}'"
        )
        LOGGER.info("=" * 80)
        LOGGER.info(f"  - P-value (Aciclicidad): {result.p_value:.6f}")
        LOGGER.info(
            f"  - Posterior Bayesiano de Aciclicidad: {result.bayesian_posterior:.4f}"
        )
        LOGGER.info(
            f"  - Intervalo de Confianza (95%%): [{result.confidence_interval[0]:.4f}, {result.confidence_interval[1]:.4f}]"
        )
        LOGGER.info(
            f"  - Poder EstadÃstico: {result.statistical_power:.4f} {'(ADECUADO)' if result.adequate_power else '(INSUFICIENTE)'}"
        )
        LOGGER.info(f"  - Score de Robustez Estructural: {result.robustness_score:.4f}")
        LOGGER.info(f"  - Tiempo de CÃ³mputo: {result.computation_time:.3f}s")
        LOGGER.info("  - Nodos validados contra schema: %d", len(serialized_nodes))
        LOGGER.info("=" * 80)


# ============================================================================
# 7. PUNTO DE ENTRADA
# ============================================================================
```