```
        FileNotFoundError: If file doesn't exist
    """
    if yaml is None:
        raise ImportError("PyYAML is required for YAML operations. Install with: pip
install pyyaml")

    file_path = Path(file_path)

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

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

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

def _is_calibration_file(path: Path) -> bool:
    stem = path.stem.lower()
    return any(keyword in stem for keyword in ("calibr", "calib", "calibracion"))

@lru_cache(maxsize=1)
def list_calibration_files() -> dict[str, Path]:
    """Return mapping of calibration name -> file path detected in search paths."""
    files: dict[str, Path] = {}
    for base in _CALIBRATION_SEARCH_PATHS:
        if not base.exists():
            continue
        for pattern in ("*.yaml", "*.yml"):
            for candidate in base.glob(pattern):
                if not candidate.is_file():
                    continue
                if not _is_calibration_file(candidate):
                    continue
                key = candidate.stem
                # Prefer higher-priority paths (earlier entries in search list)
                files.setdefault(key, candidate)
    return files

def load_calibration(name: str) -> dict[str, Any]:
    """Load a single calibration YAML by name (stem or filename).

    DEPRECATED: External YAML calibration loading is deprecated.
    Use internal calibration_registry.py for all calibrations.
    This function is maintained only for backwards compatibility.
    """
    import warnings
    warnings.warn(
        "load_calibration() is deprecated. Use calibration_registry.py for internal
calibrations.",
        DeprecationWarning,
        stacklevel=2
    )

    # Raise error to block usage - deprecated path
    raise RuntimeError(
        "Deprecated calibration path: External YAML calibration loading is no longer
supported. "
        "All calibrations must be defined in calibration_registry.py. "
        f"Attempted to load: {name}"
    )


def load_all_calibrations(include_metadata: bool = True) -> dict[str, dict[str, Any]]:
    """Load all detected calibration YAML files.

    DEPRECATED: External YAML calibration loading is deprecated.
    Use internal calibration_registry.py for all calibrations.
    This function is maintained only for backwards compatibility.

    Args:
        include_metadata: When True, attach helper metadata (path, targets) to each
calibration entry.

    Returns:
        Empty dictionary - YAML calibrations no longer supported
    """
    import logging
    import warnings

    logger = logging.getLogger(__name__)

    warnings.warn(
        "load_all_calibrations() is deprecated. Use calibration_registry.CALIBRATIONS for
internal calibrations.",
        DeprecationWarning,
        stacklevel=2
    )

    logger.warning(
        "DEPRECATED: load_all_calibrations() called. "
        "External YAML calibration loading is no longer supported. "
        "Use calibration_registry.CALIBRATIONS instead. "
        "Returning empty dict."
    )

    # Return empty dict - no YAML calibrations loaded
    return {}

# ============================================================================
# TEXT FILE I/O OPERATIONS
# ============================================================================

def read_text_file(file_path: str | Path) -> str:
    """
    Read text file with UTF-8 encoding.

    Args:
        file_path: Path to text file

    Returns:
        String content of the file

    Raises:
        FileNotFoundError: If file doesn't exist
    """
    file_path = Path(file_path)

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

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

    logger.debug(f"Read {len(content)} characters from {file_path}")
    return content

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

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

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

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

# ============================================================================
# CSV I/O OPERATIONS
# ============================================================================

def write_csv(rows: list[list[Any]], file_path: str | Path, headers: list[str] = None) ->
None:
    """
    Write data to CSV file.

    Args:
        rows: List of rows to write
        file_path: Path to output CSV file
        headers: Optional list of column headers
    """
    file_path = Path(file_path)
    file_path.parent.mkdir(parents=True, exist_ok=True)

    with open(file_path, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)

        if headers:
            writer.writerow(headers)

        writer.writerows(rows)

    logger.info(f"Written {len(rows)} rows to CSV {file_path}")

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

def open_pdf_with_fitz(file_path: str | Path):
    """
    Open a PDF file using PyMuPDF (fitz).

    Args:
        file_path: Path to PDF file

    Returns:
        fitz.Document object

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

    file_path = Path(file_path)

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

    return fitz.open(file_path)

def open_pdf_with_pdfplumber(file_path: str | Path):
    """
    Open a PDF file using pdfplumber.

    Args:
        file_path: Path to PDF file
    Returns:
        pdfplumber.PDF object

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

    file_path = Path(file_path)

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

    return pdfplumber.open(file_path)

# ============================================================================
# SPACY MODEL LOADING
# ============================================================================

def load_spacy_model(model_name: str):
    """
    Load a spaCy language model.

    Args:
        model_name: Name of the spaCy model to load

    Returns:
        Loaded spaCy Language object

    Raises:
        ImportError: If spaCy is not installed
        OSError: If model is not found
    """
    if spacy is None:
        raise ImportError("spaCy is required. Install with: pip install spacy")

    try:
        nlp = spacy.load(model_name)
        logger.info(f"Loaded spaCy model: {model_name}")
        return nlp
    except OSError:
        logger.error(f"spaCy model '{model_name}' not found. Download with: python -m
spacy download {model_name}")
        raise

===== FILE: src/saaaaaa/analysis/financiero_viabilidad_tablas.py =====
"""
MUNICIPAL DEVELOPMENT PLAN ANALYZER - PDET COLOMBIA
===================================================
Versión: 5.0 - Causal Inference Edition (2025)
Especialización: Planes de Desarrollo Municipal con Análisis Causal Bayesiano
Arquitectura: Extracción Avanzada + Inferencia Causal + DAG Learning + Counterfactuals

NUEVA CAPACIDAD - INFERENCIA CAUSAL:
✓ Identificación automática de mecanismos causales en PDM
✓ Construcción de DAGs (Directed Acyclic Graphs) para pilares PDET
✓ Estimación bayesiana de efectos causales directos e indirectos
✓ Análisis contrafactual de intervenciones
✓ Cuantificación de heterogeneidad causal por contexto territorial
✓ Detección de confounders y mediadores
✓ Análisis de sensibilidad para supuestos de identificación

COMPLIANCE:
✓ Python 3.10+ con type hints completos
✓ Sin placeholders - 100% implementado y probado
✓ Integración completa con pipeline existente
✓ 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

# === EXTRACCIÓN 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

# === ESTADÍSTICA 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 saaaaaa.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 saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

_lockdown = get_dependency_lockdown()

# ============================================================================
# LOGGING CONFIGURATION
# ============================================================================
logger = logging.getLogger(__name__)

# ============================================================================
# CONFIGURACIÓN ESPECÍFICA PARA COLOMBIA Y PDET
# ============================================================================

class ColombianMunicipalContext:
    """Contexto específico del marco normativo colombiano para PDM"""

    OFFICIAL_SYSTEMS: dict[str, str] = {
        'SISBEN': r'SISB[EÉ]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[íi]as|SGR',
        'FUT': r'Formulario\s+[ÚU]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[óo]digo\s+DANE)\s*[:\-]?\s*(\d{5,8})',
        'MGA': r'Metodolog[íi]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] = [
    'Dimensión Económica',
    'Dimensión Social',
    'Dimensión Ambiental',
    'Dimensión Institucional',
    'Dimensión Territorial'
]

PDET_PILLARS: list[str] = [
    'Ordenamiento social de la propiedad rural',
    'Infraestructura y adecuación de tierras',
    'Salud rural',
    'Educación rural y primera infancia',
    'Vivienda, agua potable y saneamiento básico',
    'Reactivación económica y producción agropecuaria',
    'Sistema para la garantía progresiva del derecho a la alimentación',
    'Reconciliación, 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 adecuación 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
    },
    'Educación rural y primera infancia': {
        'outcomes': ['cobertura_educativa', 'calidad_educativa'],
        'mediators': ['infraestructura_escolar', 'docentes_calificados'],
        'lag_years': 5
    },
    'Vivienda, agua potable y saneamiento básico': {
        'outcomes': ['deficit_habitacional', 'enfermedades_hidricas'],
        'mediators': ['cobertura_acueducto', 'viviendas_dignas'],
        'lag_years': 3
    },
    'Reactivación económica y producción agropecuaria': {
        'outcomes': ['ingreso_rural', 'empleo_rural'],
        'mediators': ['credito_rural', 'asistencia_tecnica'],
        'lag_years': 2
    },
    'Sistema para la garantía progresiva del derecho a la alimentación': {
        'outcomes': ['seguridad_alimentaria', 'nutricion_infantil'],
        'mediators': ['produccion_local', 'acceso_alimentos'],
        'lag_years': 2
    },
    'Reconciliación, convivencia y paz': {
        'outcomes': ['cohesion_social', 'confianza_institucional'],
            'mediators': ['espacios_participacion', 'justicia_transicional'],
            'lag_years': 6
        }
    }

    INDICATOR_STRUCTURE: dict[str, list[str]] = {
        'resultado': ['línea_base', 'meta', 'año_base', 'año_meta', 'fuente',
'responsable'],
        'producto': ['indicador', 'fórmula', 'unidad_medida', 'línea_base', 'meta',
'periodicidad'],
        'gestión': ['eficacia', 'eficiencia', 'economía', '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 Acíclico 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['secretaría', 'oficina', 'dirección', 'alcaldía', '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("🔧 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 .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("✓ Modelos inicializados correctamente\n")

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._get_spanish_stopwords")
    def _get_spanish_stopwords(self) -> list[str]:
        base_stopwords = spacy.lang.es.stop_words.STOP_WORDS
        gov_stopwords = {
            'artículo', 'decreto', 'mediante', 'conforme', 'respecto',
            'acuerdo', 'resolución', 'ordenanza', 'literal', 'numeral'
        }
        return list(base_stopwords | gov_stopwords)

    # ============================================================================
    # EXTRACCIÓN DE TABLAS
    # ============================================================================

    async def extract_tables(self, pdf_path: str) -> list[ExtractedTable]:
        print("📊 Iniciando extracción 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'] > get_parameter_loader().get("saaaaaa.
analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords").g
et("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"  ⚠  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'] > get_parameter_loader().get("saaaaaa.
analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords").g
et("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" ⚠  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 = get_parameter_loader().get("saaaaaa.analysis.fi
nanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords").get("confide
nce_score", 0.6) # Refactored
                    ))
        except Exception as e:
            print(f" ⚠  Tabula: {str(e)[:50]}")

        unique_tables = self._deduplicate_tables(all_tables)
        print(f"✓ {len(unique_tables)} tablas únicas extraídas\n")

        reconstructed = await self._reconstruct_fragmented_tables(unique_tables)
        print(f"🖉 {len(reconstructed)} tablas después de reconstitución\n")

        classified = self._classify_tables(reconstructed)
        return classified

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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] > get_parameter_loader().get("saaaaaa.analysis.f
inanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables").get("auto_para
m_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=get_parameter_loader().get("saaaaaa.analysis.financiero_vi
abilidad_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables").get("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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._classify_tables")
    def _classify_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
        classification_patterns = {
            'presupuesto': ['presupuesto', 'recursos', 'millones', 'sgp', 'sgr', 'fuente',
 'financiación'],
            'indicadores': ['indicador', 'línea base', 'meta', 'fórmula', 'unidad de
medida', 'periodicidad'],
            'cronograma': ['cronograma', 'actividad', 'mes', 'trimestre', 'año', 'fecha'],
            'responsables': ['responsable', 'secretaría', 'dirección', 'oficina',
'ejecutor'],
            'diagnostico': ['diagnóstico', 'problema', 'causa', 'efecto', 'situación
actual'],
            'pdet': ['pdet', 'iniciativa', 'pilar', 'patr', 'transformación 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

    # ============================================================================
    # ANÁLISIS FINANCIERO
    # ============================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.analyze_financial_feasibility")
    def analyze_financial_feasibility(self, tables: list[ExtractedTable], text: str) ->
dict[str, Any]:
        print("💰 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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=(get_parameter_loader().get("saaaaaa.analysis.
financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts").get("a
uto_param_L595_45", 0.0), get_parameter_loader().get("saaaaaa.analysis.financiero_viabilid
ad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts").get("auto_param_L595_50",
 0.0)),
                        risk_level = get_parameter_loader().get("saaaaaa.analysis.financie
ro_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts").get("risk_leve
l", 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" ✓ {len(indicators)} indicadores financieros extraídos")
        return indicators

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._identify_funding_source")
    def _identify_funding_source(self, context: str) -> str:
        sources = {
            'SGP': ['sgp', 'sistema general de participaciones'],
            'SGR': ['sgr', 'regalías', 'sistema general de regalías'],
            'Recursos Propios': ['recursos propios', 'propios', 'ingresos corrientes'],
            'Cofinanciación': ['cofinanciación', 'cofinanciado'],
            'Crédito': ['crédito', 'préstamo', 'endeudamiento'],
            'Cooperación': ['cooperación internacional', 'donación'],
            'PDET': ['pdet', 'paz', 'transformación 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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', 'financiación', '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=(get_parameter_loader().get("saaaaaa.analysis.fina
nciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table").get("auto_
param_L660_41", 0.0), get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_t
ablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table").get("auto_param_L660_46",
0.0)),
                    risk_level = get_parameter_loader().get("saaaaaa.analysis.financiero_v
iabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_budget_table").get("risk_level",
0.0) # Refactored
                ))
            except Exception:
                continue

        return indicators

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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': get_parameter_loader().get("saa
aaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._analyze_funding_sour
ces").get("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': get_parameter_loader().get("saaaaaa.analysis.financiero_via
bilidad_tablas.PDETMunicipalPlanAnalyzer._analyze_funding_sources").get("auto_param_L687_3
1", 1.0) - (diversity / np.log(max(len(source_distribution), 2)))
        }

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._assess_financial_sustainability")
    def _assess_financial_sustainability(self, indicators: list[FinancialIndicator],
                                         funding_sources: dict[str, Any]) -> float:
        if not indicators:
            return get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._assess_financial_sustainability").get("auto_param_L694_19",
0.0)

        diversity_score = min(funding_sources.get('diversity_index', 0) /
funding_sources.get('max_diversity', 1), get_parameter_loader().get("saaaaaa.analysis.fina
nciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability").get(
"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
get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPla
nAnalyzer._assess_financial_sustainability").get("auto_param_L700_90", 0.0)
        pdet_dependency = distribution.get('PDET', 0) / total if total > 0 else get_parame
ter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.
_assess_financial_sustainability").get("auto_param_L701_80", 0.0)
```

```
        pdet_risk = min(pdet_dependency * 2, get_parameter_loader().get("saaaaaa.analysis.
financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability").
get("auto_param_L702_45", 1.0))

        sustainability = (diversity_score * get_parameter_loader().get("saaaaaa.analysis.f
inanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability").g
et("auto_param_L704_44", 0.3) + own_resources * get_parameter_loader().get("saaaaaa.analys
is.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustainability
").get("auto_param_L704_66", 0.4) + (1 - pdet_risk) * get_parameter_loader().get("saaaaaa.
analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._assess_financial_sustaina
bility").get("auto_param_L704_90", 0.3))
        return float(sustainability)

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._bayesian_risk_inference")
    def _bayesian_risk_inference(self, indicators: list[FinancialIndicator],
funding_sources: dict[str, Any],
                                 sustainability: float) -> dict[str, Any]:
        print(" 🎲 Ejecutando inferencia bayesiana...")

        observed_data = {
            'n_indicators': len(indicators),
            'diversity': funding_sources.get('diversity_index', 0),
            'sustainability': sustainability,
            'dependency': funding_sources.get('dependency_risk', get_parameter_loader().ge
t("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_
inference").get("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=-
get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPla
nAnalyzer._bayesian_risk_inference").get("auto_param_L721_65", 0.3), sigma=get_parameter_l
oader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._baye
sian_risk_inference").get("auto_param_L721_76", 0.1))
            sustainability_effect = pm.Normal('sustainability_effect', mu=-
get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPla
nAnalyzer._bayesian_risk_inference").get("auto_param_L722_75", 0.4), sigma=get_parameter_l
oader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._baye
sian_risk_inference").get("auto_param_L722_86", 0.1))
            dependency_effect = pm.Normal('dependency_effect', mu=get_parameter_loader().g
et("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk
_inference").get("auto_param_L723_66", 0.5), sigma=get_parameter_loader().get("saaaaaa.ana
lysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._bayesian_risk_inference").ge
t("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" ✓ 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._interpret_risk")
    def _interpret_risk(self, risk: float) -> str:
        if risk < get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabla
s.PDETMunicipalPlanAnalyzer._interpret_risk").get("auto_param_L752_18", 0.2):
            return "Riesgo bajo - Plan financieramente robusto"
        elif risk < get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tab
las.PDETMunicipalPlanAnalyzer._interpret_risk").get("auto_param_L754_20", 0.4):
            return "Riesgo moderado-bajo - Sostenibilidad probable"
        elif risk < get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tab
las.PDETMunicipalPlanAnalyzer._interpret_risk").get("auto_param_L756_20", 0.6):
            return "Riesgo moderado - Requiere monitoreo"
        elif risk < get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tab
las.PDETMunicipalPlanAnalyzer._interpret_risk").get("auto_param_L758_20", 0.8):
            return "Riesgo alto - Vulnerabilidades significativas"
        else:
            return "Riesgo crítico - Inviabilidad financiera probable"

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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
        }

    # ========================================================================
    # IDENTIFICACIÓN DE RESPONSABLES
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.identify_responsible_entities")
    def identify_responsible_entities(self, text: str, tables: list[ExtractedTable]) ->
list[ResponsibleEntity]:
        print("👥 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" ✓ {len(scored_entities)} entidades responsables identificadas")
        return sorted(scored_entities, key=lambda x: x.specificity_score, reverse=True)

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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'] > get_
parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._extract_entities_ner").get("auto_param_L804_86", 0.7):
                        entities.append(ResponsibleEntity(
                            name=entity['word'],
                            entity_type='secretaría',
                            specificity_score=entity['score'],
                            mentioned_count=1,
                            associated_programs=[],
                            associated_indicators=[],
                            budget_allocated=None
                        ))
            except Exception:
                continue

        return entities

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._extract_entities_syntax")
    def _extract_entities_syntax(self, text: str) -> list[ResponsibleEntity]:
        entities = []
        responsibility_patterns = [

r'(?:responsable|ejecutor|encargado|a\s+cargo)[:\s]+([A-ZÁ-Ú][^\.\n]{10,100})',
            r'(?:secretar[íi]a|direcci[óo]n|oficina)\s+(?:de\s+)?([A-ZÁ-Ú][^\.\n]{5,80})',

r'([A-ZÁ-Ú][^\.\n]{10,100})\s+(?:ser[áa]|estar[áa]|tendr[áa])\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=get_parameter_loader().get("saaaaaa.analysis.financi
ero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_entities_syntax").get("auto_param
_L838_38", 0.6),
                    mentioned_count=1,
                    associated_programs=[],
                    associated_indicators=[],
                    budget_allocated=None
                ))

        return entities

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._classify_entity_type")
    def _classify_entity_type(self, name: str) -> str:
        name_lower = name.lower()
        if 'secretaría' in name_lower or 'secretaria' in name_lower:
            return 'secretaría'
        elif 'dirección' in name_lower:
            return 'dirección'
        elif 'oficina' in name_lower:
            return 'oficina'
        elif 'alcaldía' in name_lower or 'alcalde' in name_lower:
            return 'alcaldía'
        else:
            return 'externo'

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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=get_parameter_loader().get("saaaaaa.analysis.financi
ero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables").get(
"auto_param_L884_38", 0.8),
                    mentioned_count=1,
                    associated_programs=[],
                    associated_indicators=[],
                    budget_allocated=None
                ))

        return entities

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 = get_parameter_loader().get("saaaaaa.analysis.financiero_via
bilidad_tablas.PDETMunicipalPlanAnalyzer._consolidate_entities").get("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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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, get_parameter_loader().get("
saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_spec
ificity").get("auto_param_L934_62", 1.0))
            propn_count = sum(1 for token in doc if token.pos_ == 'PROPN')
            propn_score = min(propn_count / 3, get_parameter_loader().get("saaaaaa.analysi
s.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity").get("
auto_param_L936_47", 1.0))

            institutional_words = ['secretaría', 'dirección', 'oficina', 'departamento',
'coordinación', 'gerencia',
                                   'subdirección']
            inst_score = float(any(word in entity.name.lower() for word in
institutional_words))
            mention_score = min(entity.mentioned_count / 10, get_parameter_loader().get("s
aaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_speci
ficity").get("auto_param_L941_61", 1.0))

            final_score = (length_score * get_parameter_loader().get("saaaaaa.analysis.fin
anciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity").get("auto_
param_L943_42", 0.2) + propn_score * get_parameter_loader().get("saaaaaa.analysis.financie
ro_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity").get("auto_param
_L943_62", 0.3) + inst_score * get_parameter_loader().get("saaaaaa.analysis.financiero_via
bilidad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity").get("auto_param_L943_
81", 0.3) + mention_score * get_parameter_loader().get("saaaaaa.analysis.financiero_viabil
idad_tablas.PDETMunicipalPlanAnalyzer._score_entity_specificity").get("auto_param_L943_103
", 0.2))

            entity.specificity_score = final_score
            scored.append(entity)

        return scored

    # ==========================================================================
    # INFERENCIA CAUSAL - DAG CONSTRUCTION
    # ==========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.construct_causal_dag")
    def construct_causal_dag(self, text: str, tables: list[ExtractedTable],
                             financial_analysis: dict[str, Any]) -> CausalDAG:
        print("🔗 Construyendo grafo causal (DAG)...")

        nodes = self._identify_causal_nodes(text, tables, financial_analysis)
        print(f" ✓ {len(nodes)} nodos causales identificados")

        edges = self._identify_causal_edges(text, nodes)
        print(f" ✓ {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 get
```

```
_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAn
alyzer.construct_causal_dag").get("auto_param_L969_87", 0.0),
                'evidence': node.evidence_strength
            })

        for edge in edges:
            if edge.probability > get_parameter_loader().get("saaaaaa.analysis.financiero_
viabilidad_tablas.PDETMunicipalPlanAnalyzer.construct_causal_dag").get("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(" ⚠  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" ✓ 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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, get_parameter_loader().get("s
aaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_no
des").get("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, get_parameter_loa
der().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identi
fy_causal_nodes").get("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, get_parameter_lo
ader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._ident
ify_causal_nodes").get("auto_param_L1043_74", 1.0))
                    )

        return nodes

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 > get_parameter_loader().get("saaaaaa.analysis.financiero_viabil
idad_tablas.PDETMunicipalPlanAnalyzer._find_semantic_mentions").get("auto_param_L1062_28",
 0.5):
                mentions.append(sentence)

        return mentions

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._find_outcome_mentions")
    def _find_outcome_mentions(self, text: str, outcome: str) -> list[str]:
        outcome_keywords = {
            'seguridad_juridica': ['seguridad jurídica', 'formalización', 'títulos',
'propiedad'],
            'reduccion_conflictos_tierra': ['conflicto', 'tierra', 'disputa',
'territorial'],
            'conectividad': ['conectividad', 'vías', 'acceso', 'transporte'],
            'productividad_agricola': ['productividad', 'agrícola', 'producción',
'rendimiento'],
            'mortalidad_infantil': ['mortalidad infantil', 'niños', 'salud infantil'],
            'esperanza_vida': ['esperanza de vida', 'longevidad', 'salud'],
            'cobertura_educativa': ['cobertura educativa', 'acceso educación',
'matrícula'],
            'calidad_educativa': ['calidad educativa', 'aprendizaje', 'pruebas saber'],
            'deficit_habitacional': ['déficit habitacional', 'vivienda', 'hogares'],
            'enfermedades_hidricas': ['enfermedades hídricas', 'agua potable',
'saneamiento'],
            'ingreso_rural': ['ingreso rural', 'pobreza rural', 'economía campesina'],
            'empleo_rural': ['empleo rural', 'trabajo campo', 'ocupación'],
            'seguridad_alimentaria': ['seguridad alimentaria', 'hambre', 'nutrición'],
            'nutricion_infantil': ['nutrición infantil', 'desnutrición', 'alimentación
niños'],
            'cohesion_social': ['cohesión 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._find_mediator_mentions")
    def _find_mediator_mentions(self, text: str, mediator: str) -> list[str]:
        mediator_patterns = {
            'formalizacion': ['formalización', 'titulación', 'escrituras'],
            'acceso_justicia': ['acceso justicia', 'juzgados', 'defensoría'],
            'vias_terciarias': ['vías terciarias', 'caminos', 'carreteras'],
            'distritos_riego': ['distritos riego', 'irrigación', 'agua agrícola'],
            'cobertura_salud': ['cobertura salud', 'eps', 'atención médica'],
            'infraestructura_salud': ['hospital', 'centro salud', 'puesto salud'],
            'infraestructura_escolar': ['escuela', 'colegio', 'infraestructura
educativa'],
            'docentes_calificados': ['docentes', 'maestros', 'profesores'],
            'cobertura_acueducto': ['acueducto', 'agua potable', 'tubería'],
            'viviendas_dignas': ['vivienda digna', 'casa', 'hogar'],
            'credito_rural': ['crédito rural', 'financiamiento', 'banco agrario'],
            'asistencia_tecnica': ['asistencia técnica', 'extensión rural', 'asesoría'],
            'produccion_local': ['producción local', 'cultivos', 'agricultura'],
            'acceso_alimentos': ['acceso alimentos', 'mercado', 'distribución'],
            'espacios_participacion': ['participación', 'comités', 'juntas'],
            'justicia_transicional': ['justicia transicional', 'víctimas', 'reparación']
        }

        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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 según teoría PDET",
                        probability = get_parameter_loader().get("saaaaaa.analysis.financi
ero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges").get("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 = get_parameter_loader().get("saaaaaa.analysis
.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges").get("prob
ability", 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)

                    if existing:
                        existing.probability = min(existing.probability + get_parameter_lo
ader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._ident
ify_causal_edges").get("auto_param_L1201_74", 0.2), get_parameter_loader().get("saaaaaa.an
alysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges").get
("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 = get_parameter_loader().get("saaaaaa.analysis.fin
anciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._identify_causal_edges").get("probabil
ity", 0.6) # Refactored
                        ))

        edges = self._refine_edge_probabilities(edges, text, nodes)

        return edges

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 = get_parameter_loader().get("saaaaaa.analysis.financiero_viabilid
ad_tablas.PDETMunicipalPlanAnalyzer._match_text_to_node").get("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 > get_parameter_loader().get("s
aaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._match_text_to_node
").get("auto_param_L1235_61", 0.4):
                best_similarity = similarity
                best_match = node_name

        return best_match

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 * get_parameter_loader().get("saaaaaa.analy
sis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities").ge
t("auto_param_L1260_49", 0.1), get_parameter_loader().get("saaaaaa.analysis.financiero_via
bilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_probabilities").get("auto_param_L126
0_54", 0.3))
                edge.probability = min(edge.probability + boost, get_parameter_loader().ge
t("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._refine_edge_pr
obabilities").get("auto_param_L1261_65", 1.0))

        return edges

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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', g
et_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlan
Analyzer._break_cycles").get("auto_param_L1270_89", 0.5)))
                G.remove_edge(weakest_edge[0], weakest_edge[1])
            except nx.NetworkXNoCycle:
                break

        return G

    # ==========================================================================
    # ESTIMACIÓN BAYESIANA DE EFECTOS CAUSALES
    # ==========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.estimate_causal_effects")
    def estimate_causal_effects(self, dag: CausalDAG, text: str, financial_analysis:
dict[str, Any]) -> list[
        CausalEffect]:
        print("📈 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" ✓ {len(effects)} efectos causales estimados")
        return effects

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 get_parameter_loader().get("saaaaaa.analysis.financi
ero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian").get("auto_para
m_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 * get_para
meter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyze
r._estimate_effect_bayesian").get("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 * get_parameter_l
oader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._esti
mate_effect_bayesian").get("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=get_parameter_loader().get("saaaa
aa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesi
an").get("auto_param_L1350_57", 0.5))

            pm.Normal('pseudo_obs', mu=total_effect, sigma=obs_noise,
                      observed=np.array([evidence_strength * get_parameter_lo
ader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estim
ate_effect_bayesian").get("auto_param_L1353_74", 0.5)]))

            trace = pm.sample(1500, tune=800, cores=1, return_inferencedata=True,
progressbar=False, target_accept=get_parameter_loader().get("saaaaaa.analysis.financiero_v
iabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian").get("auto_param_L13
55_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) > get_parameter_loader().ge
t("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effec
t_bayesian").get("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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._get_prior_effect")
    def _get_prior_effect(self, treatment: str, outcome: str) -> tuple[float, float]:
        """
        Priors informados basados en meta-análisis de programas PDET
        Referencia: Cinelli et al. (2022) - Sensitivity Analysis for Causal Inference
        """
        effect_priors = {
            ('Infraestructura y adecuación de tierras', 'productividad_agricola'): (get_pa
rameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnaly
zer._get_prior_effect").get("auto_param_L1385_84", 0.35), get_parameter_loader().get("saaa
aaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect").ge
t("auto_param_L1385_90", 0.15)),
            ('Salud rural', 'mortalidad_infantil'): (-
get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPla
nAnalyzer._get_prior_effect").get("auto_param_L1386_54", 0.28), get_parameter_loader().get
("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effec
t").get("auto_param_L1386_60", 0.12)),
            ('Educación rural y primera infancia', 'cobertura_educativa'): (get_parameter_
loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get
_prior_effect").get("auto_param_L1387_76", 0.42), get_parameter_loader().get("saaaaaa.anal
ysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect").get("auto_
param_L1387_82", 0.18)),
            ('Vivienda, agua potable y saneamiento básico', 'enfermedades_hidricas'): (-
get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPla
nAnalyzer._get_prior_effect").get("auto_param_L1388_88", 0.33), get_parameter_loader().get
("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effec
t").get("auto_param_L1388_94", 0.14)),
            ('Reactivación económica y producción agropecuaria', 'ingreso_rural'): (get_pa
rameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnaly
zer._get_prior_effect").get("auto_param_L1389_84", 0.29), get_parameter_loader().get("saaa
aaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect").ge
t("auto_param_L1389_90", 0.16)),
            ('Sistema para la garantía progresiva del derecho a la alimentación',
'seguridad_alimentaria'): (get_parameter_loader().get("saaaaaa.analysis.financiero_viabili
dad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect").get("auto_param_L1390_109",
0.38),

                  get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabla
s.PDETMunicipalPlanAnalyzer._get_prior_effect").get("auto_param_L1391_108", 0.17)),
        }

        if (treatment, outcome) in effect_priors:
            return effect_priors[(treatment, outcome)]

        return (get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.
PDETMunicipalPlanAnalyzer._get_prior_effect").get("auto_param_L1397_16", 0.2), get_paramet
er_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._
get_prior_effect").get("auto_param_L1397_21", 0.25))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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

    # ========================================================================
    # ANÁLISIS CONTRAFACTUAL (Pearl's Three-Layer Causal Hierarchy)
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.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?"

        Implementación basada en:
        - Pearl & Mackenzie (2018) - The Book of Why
        - Sharma & Kiciman (2020) - DoWhy: An End-to-End Library for Causal Inference
        """
        print("🔮 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 get_parameter_loader().get("saaaaaa.analysis.financ
iero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_counterfactuals").get("auto_para
m_L1438_101", 0.0)
            for node in pillar_nodes
        }
        total_budget = sum(current_budgets.values())

        if total_budget == 0:
            print(" ⚠  No hay información 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 educación y salud
        priority_pillars = ['Educación 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,
                                                  "Priorización educación y salud")
        scenarios.append(scenario_2)

        # Escenario 3: Focalización 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 * get_parameter_loader().get("saaaaaa.analysis.
financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_counterfactuals").get("aut
o_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"Focalización en
{best_pillar[:40]}")
            scenarios.append(scenario_3)

        print(f" ✓ {len(scenarios)} escenarios contrafactuales generados")
        return scenarios

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._simulate_intervention")
    def _simulate_intervention(self, intervention: dict[str, float], dag: CausalDAG,
                               causal_effects: list[CausalEffect], description: str) ->
CounterfactualScenario:
        """
        Simula intervención usando do-calculus (Pearl, 2009)
        Implementa: P(Y | do(X=x)) mediante propagación 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 = get_parameter_loader().get("saaaaaa.analysis.financiero_viab
ilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention").get("expected_change",
0.0) # Refactored
            variance_sum = get_parameter_loader().get("saaaaaa.analysis.financiero_viabili
dad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention").get("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 get_parameter_loader().get("saaaaaa.
analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simulate_intervention").g
et("auto_param_L1512_54", 0.0)
                new_budget = intervention[treatment]

                budget_multiplier = new_budget / current_budget if current_budget > 0 else
 get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPl
anAnalyzer._simulate_intervention").get("auto_param_L1515_91", 1.0)

                # Rendimientos decrecientes: log transform
                effect_multiplier = np.log1p(budget_multiplier) / np.log1p(get_parameter_l
oader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._simu
late_intervention").get("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 ≈ 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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 += "**Intervención 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 > get_parame
ter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.
_generate_scenario_narrative").get("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

    # ========================================================================
    # ANÁLISIS DE SENSIBILIDAD (Cinelli et al., 2022)
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.sensitivity_analysis")
    def sensitivity_analysis(self, causal_effects: list[CausalEffect], dag: CausalDAG) ->
dict[str, Any]:
        """
        Análisis de sensibilidad para supuestos de identificación causal
        Basado en: Cinelli, Forney & Pearl (2022) - "A Crash Course in Good and Bad
Controls"
        """
        print("🔬 Ejecutando análisis 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]}→{effect.outcome[:30]}"] = {
                'e_value': unobserved_confounding,
                'robustness_value': robustness_value,
                'interpretation': self._interpret_sensitivity(unobserved_confounding,
robustness_value)
            }

        print(f" ✓ Sensibilidad analizada para {len(sensitivity_results)} efectos")
        return sensitivity_results

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._compute_e_value")
    def _compute_e_value(self, effect: CausalEffect) -> float:
        """
        E-value: mínima fuerza de confounding no observado para anular el efecto
        Fórmula: E = effect_estimate + sqrt(effect_estimate * (effect_estimate - 1))

        Referencia: VanderWeele & Ding (2017) - Ann Intern Med
        """
        if effect.posterior_mean <= 0:
            return get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._compute_e_value").get("auto_param_L1612_19", 1.0)

        rr = np.exp(effect.posterior_mean)  # Convert log-scale to risk ratio
        if rr <= 1:
            return get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._compute_e_value").get("auto_param_L1616_19", 1.0)
        e_value = rr + np.sqrt(rr * (rr - 1))

        return float(e_value)

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._compute_robustness_value")
    def _compute_robustness_value(self, effect: CausalEffect, dag: CausalDAG) -> float:
        """
        Robustness Value: percentil de la distribución posterior que cruza cero
        Valores altos (>get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad
_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value").get("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 get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._compute_robustness_value").get("auto_param_L1630_19", 1.0)

        width = ci_upper - ci_lower
        if width == 0:
            return get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._compute_robustness_value").get("auto_param_L1634_19", 0.5)

        robustness = abs(effect.posterior_mean) / (width / 2)
        return float(min(robustness, get_parameter_loader().get("saaaaaa.analysis.financie
ro_viabilidad_tablas.PDETMunicipalPlanAnalyzer._compute_robustness_value").get("auto_param
_L1637_37", 1.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._interpret_sensitivity")
    def _interpret_sensitivity(self, e_value: float, robustness: float) -> str:
        """Interpretación de resultados de sensibilidad"""

        if e_value > 2.0 and robustness > get_parameter_loader().get("saaaaaa.analysis.fin
anciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity").get("auto_par
am_L1643_42", 0.8):
            return "Efecto robusto - Resistente a confounding no observado"
        elif e_value > 1.5 and robustness > get_parameter_loader().get("saaaaaa.analysis.f
inanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity").get("auto_p
aram_L1645_44", 0.6):
            return "Efecto moderadamente robusto - Precaución con confounders"
        elif e_value > 1.2 and robustness > get_parameter_loader().get("saaaaaa.analysis.f
inanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._interpret_sensitivity").get("auto_p
aram_L1647_44", 0.4):
            return "Efecto sensible - Alta vulnerabilidad a confounding"
        else:
            return "Efecto frágil - Resultados no confiables sin ajustes adicionales"

    # ============================================================================
    # SCORING INTEGRAL DE CALIDAD
    # ============================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.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 análisis con pesos calibrados
        """
        print("⭐ 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([get_parameter_loader().get("saaaaaa.analysis.financiero_viabil
idad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score").get("auto_param_L1675_28",
 0.20), get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMuni
cipalPlanAnalyzer.calculate_quality_score").get("auto_param_L1675_34", 0.15), get_paramete
r_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.ca
lculate_quality_score").get("auto_param_L1675_40", 0.15), get_parameter_loader().get("saaa
aaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_scor
e").get("auto_param_L1675_46", 0.10), get_parameter_loader().get("saaaaaa.analysis.financi
ero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score").get("auto_param_
L1675_52", 0.20), get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabla
s.PDETMunicipalPlanAnalyzer.calculate_quality_score").get("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" ✓ Score final: {overall_score:.2f}/1get_parameter_loader().get("saaaaaa.a
nalysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score").g
et("auto_param_L1694_53", 0.0)")

        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
        )

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tabl
as.PDETMunicipalPlanAnalyzer._score_financial_component").get("auto_param_L1714_19", 0.0)

        budget_score = min(np.log10(float(budget)) / 12, get_parameter_loader().get("saaaa
aa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_compon
ent").get("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, get_parameter_loader().get("saaa
aaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_financial_compo
nent").get("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', get_parameter_loade
r().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_fi
nancial_component").get("auto_param_L1725_71", 0.5))
        risk_score = (1 - risk) * 1.5
        return float(min(budget_score + diversity_score + sustainability_score +
risk_score, 1get_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDE
TMunicipalPlanAnalyzer._score_financial_component").get("auto_param_L1728_94", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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[íi]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 = get_parameter_loader().get("saaaaaa.analysis.financiero_viabi
lidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators").get("completeness_score", 0.0)
# Refactored
        for table in indicator_tables:
            df = table.df
            required_cols = ['indicador', 'línea 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|hectáreas)',  # 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, get_parameter_loader().get("saaaaaa.analysis.f
inanciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_indicators").get("auto_param_
L1760_44", 1.0)) * 3.0

        formula_mentions = len(re.findall(r'f[óo]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, get_paramete
r_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._s
core_indicators").get("auto_param_L1765_78", 1.0)) * 3.0

        return float(min(completeness_score + smart_score + technical_score, 1get_paramete
r_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._s
core_indicators").get("auto_param_L1767_78", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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, get_parameter_loader().get("saaaaaa.analysis
.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_responsibility_clarity").ge
t("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 ['secretaría',
'dirección', '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, 1get_param
eter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer
._score_responsibility_clarity").get("auto_param_L1785_81", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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, get_parameter_loader().get("saaaaaa.analysis.fin
anciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_temporal_consistency").get("aut
o_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', 'año', '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, get_parameter_loader().get("saaaaaa.analysis.fin
anciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_temporal_consistency").get("aut
o_param_L1805_42", 1.0)) * 3.0

        return float(min(range_score + cronograma_score + term_score, 1get_parameter_loade
r().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_te
mporal_consistency").get("auto_param_L1807_71", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._score_pdet_alignment")
    def _score_pdet_alignment(self, text: str, tables: list[ExtractedTable], dag:
CausalDAG) -> float:
        """Score alineación 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, get_parameter_loader().
get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_pdet_a
lignment").get("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, 1get_parameter_loa
der().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._score_
pdet_alignment").get("auto_param_L1833_73", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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), get_param
eter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer
._score_causal_coherence").get("auto_param_L1844_81", 1.0)) * 3.0

        if not effects:
            effect_quality = get_parameter_loader().get("saaaaaa.analysis.financiero_viabi
lidad_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence").get("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, 1get_parameter_l
oader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._scor
e_causal_coherence").get("auto_param_L1858_75", 0.0)))

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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, get_parameter_loader().get("saaaaaa.analysis.finan
ciero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_score_confidence").get("auto_p
aram_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))

    # ============================================================================
    # EXPORTACIÓN Y VISUALIZACIÓN
    # ==========================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.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', get_parameter_loader().get("saaaaaa.analys
is.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.export_causal_network").get("aut
o_param_L1891_48", 0.0))

        for _u, _v, data in G.edges(data=True):
            data['weight'] = data.get('probability', get_parameter_loader().get("saaaaaa.a
nalysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.export_causal_network").get
("auto_param_L1894_53", 0.5))
            data['edge_type'] = data.get('type', 'unknown')

        nx.write_graphml(G, output_path)
        print(f"✓ Red causal exportada a: {output_path}")

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.generate_executive_report")
    def generate_executive_report(self, analysis_results: dict[str, Any]) -> str:
        """Genera reporte ejecutivo en Markdown"""

        report = "# ANÁLISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET\n\n"
        report += f"**Fecha de análisis:** {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}/1get_param
eter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer
.generate_executive_report").get("auto_param_L1910_81", 0.0) "
        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"- **Alineación PDET:** {quality['pdet_alignment']:.1f}/10\n"
        report += f"- **Coherencia Causal:** {quality['causal_coherence']:.1f}/10\n\n"

        report += "## 2. ANÁLISIS FINANCIERO\n\n"
        fin = analysis_results['financial_analysis']
        report += f"**Presupuesto Total:** ${fin['total_budget']:,.0f} COP\n\n"

        report += "### Distribución 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**Índice de Diversificación:**
{fin['funding_sources'].get('diversity_index', 0):.2f}\n"
        report += f"**Score de Sostenibilidad:** {fin['sustainability_score']:.2f}\n"
        report += f"**Evaluación 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'] > ge
t_parameter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanA
nalyzer.generate_executive_report").get("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]} → {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"- Vías de mediación: {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. ANÁLISIS DE SENSIBILIDAD\n\n"

        sensitivity = analysis_results.get('sensitivity_analysis', {})
        if sensitivity:
            report += "| Relación Causal | E-Value | Robustez | Interpretación |\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 += "*Análisis generado por PDETMunicipalPlanAnalyzer v5.0*\n"
        report += "*Metodología: Inferencia Causal Bayesiana + Structural Causal
Models*\n"

        return report

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._interpret_overall_quality")
    def _interpret_overall_quality(self, score: float) -> str:
        """Interpretación del score global"""

        if score >= 8.0:
            return ("**Evaluación: EXCELENTE** ✓\n\n"
                    "El plan cumple con altos estándares de calidad técnica. "
                    "Presenta coherencia causal sólida, viabilidad financiera demostrable,
 "
```

```
                    "y alineación robusta con los pilares PDET.")
        elif score >= 6.5:
            return ("**Evaluación: BUENO** ✓\n\n"
                    "El plan presenta bases sólidas pero con oportunidades de mejora. "
                    "Se recomienda fortalecer algunos componentes específicos.")
        elif score >= 5.0:
            return ("**Evaluación: ACEPTABLE** ⚠ \n\n"
                    "El plan cumple requisitos mínimos pero requiere ajustes sustanciales
"
                    "en múltiples dimensiones para asegurar efectividad.")
        else:
            return ("**Evaluación: DEFICIENTE** ✕ \n\n"
                    "El plan presenta deficiencias críticas que comprometen su viabilidad.
 "
                    "Se requiere reformulación integral.")

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._generate_recommendations")
    def _generate_recommendations(self, analysis_results: dict[str, Any]) -> str:
        """Genera recomendaciones específicas basadas en el análisis"""

        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) > get_parameter_loader().g
et("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_reco
mmendations").get("auto_param_L2015_66", 0.6):
                recommendations.append(
                    "**Diversificación de fuentes:** Reducir dependencia excesiva de
fuentes únicas. "
                    "Explorar alternativas como cooperación internacional, APP, o gestión
de recursos propios."
                )

            if fin['sustainability_score'] < get_parameter_loader().get("saaaaaa.analysis.
financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations").get("au
to_param_L2021_45", 0.5):
                recommendations.append(
                    "**Sostenibilidad fiscal:** Fortalecer componente de recursos propios.
 "
                    "Desarrollar estrategias de generación de ingresos municipales."
                )

        # Recomendaciones de indicadores
        if quality['indicator_quality'] < 6.0:
            recommendations.append(
                "**Fortalecimiento de indicadores:** Definir indicadores SMART completos "
                "(específicos, medibles, alcanzables, relevantes, temporales) con líneas
base, "
                "metas cuantificadas, fórmulas de cálculo y fuentes verificables."
            )

        # Recomendaciones causales
        effects = analysis_results.get('causal_effects', [])
        if effects:
            weak_effects = [e for e in effects if e['probability_significant'] < get_param
eter_loader().get("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer
._generate_recommendations").get("auto_param_L2038_81", 0.5)]

            if len(weak_effects) > len(effects) * get_parameter_loader().get("saaaaaa.anal
ysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._generate_recommendations").ge
t("auto_param_L2040_50", 0.5):
                recommendations.append(
                    "**Robustez causal:** Fortalecer vínculos entre intervenciones y
resultados esperados. "
```

```
                "Explicitar teorías de cambio y mecanismos causales subyacentes."
            )

        # Recomendaciones PDET
        if quality['pdet_alignment'] < 6.0:
            recommendations.append(
                "**Alineación PDET:** Articular explícitamente con los 8 pilares del Pacto
 Estructurante. "
                "Referenciar iniciativas PATR y asegurar coherencia con transformación
territorial."
            )

        # Recomendaciones de responsabilidad
        if quality['responsibility_clarity'] < 6.0:
            recommendations.append(
                "**Claridad institucional:** Especificar responsables concretos para cada
programa. "
                "Evitar asignaciones genéricas como 'todas las secretarías' o 'alcaldía
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"**Optimización 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 implementación según lo planificado.\n"

        result = ""
        for i, rec in enumerate(recommendations, 1):
            result += f"{i}. {rec}\n\n"

        return result

    # ============================================================================
    # PIPELINE PRINCIPAL
    # ============================================================================

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.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 análisis

        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 análisis
        """

        print("\n" + "=" * 70)
        print("ANÁLISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET")
        print("=" * 70 + "\n")

        start_time = datetime.now()

        # 1. Extracción de texto
        print("🗐 Extrayendo texto del PDF...")
        full_text = self._extract_full_text(pdf_path)
        print(f" ✓ {len(full_text)} caracteres extraídos\n")

        # 2. Extracción de tablas
        tables = await self.extract_tables(pdf_path)

        # 3. Análisis financiero
        financial_analysis = self.analyze_financial_feasibility(tables, full_text)

        # 4. Identificación de responsables
        responsible_entities = self.identify_responsible_entities(full_text, tables)

        # 5. Construcción de DAG causal
        causal_dag = self.construct_causal_dag(full_text, tables, financial_analysis)

        # 6. Estimación de efectos causales
        causal_effects = self.estimate_causal_effects(causal_dag, full_text,
financial_analysis)

        # 7. Generación de contrafactuales
        counterfactuals = self.generate_counterfactuals(causal_dag, causal_effects,
financial_analysis)

        # 8. Análisis 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. Compilación 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. Exportación 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 .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"✓ Reporte ejecutivo guardado en: {report_path}")

            # Exportar JSON
            json_path = output_path / "analysis_results.json"
            save_json(results, json_path)
            print(f"✓ Resultados JSON guardados en: {json_path}")

        elapsed = (datetime.now() - start_time).total_seconds()
        print(f"\n⏱️  Análisis completado en {elapsed:.1f} segundos")
        print("=" * 70 + "\n")

        return results

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._extract_full_text")
    def _extract_full_text(self, pdf_path: str) -> str:
        """Extrae texto completo del PDF usando múltiples métodos"""

        text_parts = []

        # Método 1: PyMuPDF (rápido y eficiente)
        # Delegate to factory for I/O operation
        from .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" ⚠  PyMuPDF falló: {str(e)[:50]}")

        # Método 2: pdfplumber (mejor para tablas complejas)
        try:
            pdf = open_pdf_with_pdfplumber(pdf_path)
            for page in pdf.pages[:100]:  # Límite de 100 páginas
                text = page.extract_text()
                if text:
                    text_parts.append(text)
            pdf.close()
        except Exception as e:
            print(f" ⚠  pdfplumber falló: {str(e)[:50]}")
        full_text = '\n\n'.join(text_parts)

        # Limpieza básica
        full_text = re.sub(r'\n{3,}', '\n\n', full_text)
        full_text = re.sub(r' {2,}', ' ', full_text)

        return full_text

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer._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 línea base cuantitativa basada
 en diagnóstico 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 específica"
            elif gap_type == 'missing_budget':
                remediation['recommendation'] = "Asignar presupuesto específico con fuente
 de financiación"
            elif gap_type == 'missing_indicator':
                remediation['recommendation'] = "Definir indicador medible con fórmula de
cálculo"
            else:
                remediation['recommendation'] = f"Completar {gap_type} según estándares
DNP"

            remediations.append(remediation)

        return remediations

    @calibrated_method("saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAna
lyzer.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) < get_parameter_loader().get("
saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommend
ations").get("auto_param_L2379_62", 0.7):
            recommendations.append(
                "Revisar sostenibilidad financiera y diversificar fuentes de financiación"
            )

        # Check indicator quality
        if analysis_results.get('indicator_quality', 0) < get_parameter_loader().get("saaa
aaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendatio
ns").get("auto_param_L2385_58", 0.7):
            recommendations.append(
                "Mejorar calidad de indicadores: asegurar línea base, meta y fuente de
información"
            )

        # Check responsibility clarity
        if analysis_results.get('responsibility_clarity', 0) < get_parameter_loader().get(
"saaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommen
dations").get("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) < get_parameter_loader().get("s
aaaaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommenda
tions").get("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) < get_parameter_loader().get("saaaa
aa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendation
s").get("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) < get_parameter_loader().get("saa
aaaa.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendati
ons").get("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 estándares mínimos. Considerar monitoreo continuo."
            )

        return recommendations

# ============================================================================
# UTILIDADES Y HELPERS
# ============================================================================

class PDETAnalysisException(Exception):
    """Excepción personalizada para errores de análisis"""
    pass

def validate_pdf_path(pdf_path: str) -> Path:
    """Valida que el path del PDF exista y sea válido"""

    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 análisis"""

    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 = get_parameter_loader().get("saaaaaa.analysis.financiero_via
bilidad_tablas.PDETMunicipalPlanAnalyzer.generate_recommendations").get("confidence_thresh
old", 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 análisis completo
        results = await analyzer.analyze_municipal_plan(
            pdf_path=pdf_path,
            output_dir="outputs/analisis_pdm"
        )

        # Acceder a resultados específicos
        print("\n📊 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"❌ Error de análisis: {e}")
    except Exception as e:
        print(f"❌ Error inesperado: {e}")
        raise

===== FILE: src/saaaaaa/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 saaaaaa.core.runtime_config import RuntimeConfig, get_runtime_config
from saaaaaa.core.contracts.runtime_contracts import (
    GraphMetricsInfo,
    FallbackCategory,
)
from saaaaaa.core.observability.structured_logging import log_fallback
from saaaaaa.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/saaaaaa/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 micro↔meso↔macro 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
```

```
from dataclasses import asdict, dataclass, field
from typing import Any

# Import runtime error fixes for defensive programming
from saaaaaa.utils.runtime_error_fixes import ensure_list_return
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

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 micro↔meso alignment
    meso_macro_alignment: float  # 0.0-1.0 meso↔macro 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"""
    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)

@dataclass
class PeerNormalization:
    """Output from Peer Normalization & Confidence Scaling"""
    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)

# ============================================================================
# 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 < τ en alguna dimensión crítica (D3, D6) → degradar
global_confidence
    - Simular impacto si se completara el cluster faltante (predictive uplift)

    OUTPUT:
    JSON {coverage_index, degraded_confidence, predictive_uplift}
    """

    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)
        """
        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 micro↔meso↔macro.

    INPUTS:
    - micro_claims (extraído de MicroLevelAnswer.evidence)
    - meso_summary_signals
    - macro_narratives (borrador)

    MANDATES:
    - Alinear claims por entidad/tema/dimensión
    - Marcar contradicción si macro afirma X y ≥k micro niegan X con posterior ≥ θ
    - Sugerir corrección: "rephrase / downgrade confidence / request re-execution"

    OUTPUT:
    JSON {contradictions[], suggested_actions}
    """

    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
        """
        self.k = contradiction_threshold
        self.theta = posterior_threshold
        logger.info(f"ContradictionScanner initialized (k={self.k}, θ={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 jerárquicos (coverage, dispersion estructural, contradictions)
    - Recalcular posterior_global y varianza

    OUTPUT:
    JSON {prior_global, penalties_applied, posterior_global, var_global}
    """

    def __init__(
        self,
        default_variance: float = 0.05
    ) -> None:
        """
        Initialize Bayesian Portfolio Composer

        Args:
            default_variance: Default variance for uncertain estimates
        """
        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 ≈ 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 0–3m / 3–6m / 6–12m 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 mínima sin colisión de prerequisitos
    - Estimar uplift esperado por tramo

    OUTPUT:
    JSON roadmap {phase, actions[], expected_uplift}
    """

    def __init__(self) -> None:
        """Initialize Roadmap Optimizer"""
        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 (get_parameter_loader().get("saaaaaa.
analysis.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L702_54", 0.0)-
get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOptimizer.__init__").get
("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, get_parameter_loader().get("saaaaaa.anal
ysis.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L752_50", 1.0))
            impact = impact_scores.get(gap_id, get_parameter_loader().get("saaaaaa.analysi
s.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L753_47", 0.5))
```

```
            # Calculate ratio (avoid division by zero)
            ratio = impact / max(effort, get_parameter_loader().get("saaaaaa.analysis.macr
o_prompts.RoadmapOptimizer.__init__").get("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": get_parameter_loader().get("saaaaaa.
analysis.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L778_54", 0.0),
"max_effort": 9.0},
            {"name": "3-6m", "actions": [], "effort": get_parameter_loader().get("saaaaaa.
analysis.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L779_54", 0.0),
"max_effort": 9.0},
            {"name": "6-12m", "actions": [], "effort": get_parameter_loader().get("saaaaaa
.analysis.macro_prompts.RoadmapOptimizer.__init__").get("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", get_parameter_loader().get("saaaaaa.analysis.ma
cro_prompts.RoadmapOptimizer.__init__").get("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 = get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOptimize
r.__init__").get("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, get_parameter_loader().get("saaaaaa.ana
lysis.macro_prompts.RoadmapOptimizer.__init__").get("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 = get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOp
timizer.__init__").get("best_impact", 0.0) # Refactored

        for endpoint in endpoints:
            path = self._trace_path(endpoint, dependency_graph)
            path_impact = sum(impact_scores.get(gap_id, get_parameter_loader().get("saaaaa
a.analysis.macro_prompts.RoadmapOptimizer.__init__").get("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 clasificación macro considerando comparativos regionales.

    INPUTS:
    - convergence_by_policy_area
    - peer_distributions: {policy_area -> {mean, std}}
    - baseline_confidence

    MANDATES:
    - Calcular z-scores
    - Penalizar si >k áreas están < -get_parameter_loader().get("saaaaaa.analysis.macro_pr
ompts.RoadmapOptimizer.__init__").get("auto_param_L956_37", 1.0) z
    - Aumentar confianza si todas dentro ±get_parameter_loader().get("saaaaaa.analysis.mac
ro_prompts.RoadmapOptimizer.__init__").get("auto_param_L957_42", 0.5) z y dispersión baja

    OUTPUT:
    JSON {z_scores, adjusted_confidence}
    """

    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
        """
        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 < -get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOpti
mizer.__init__").get("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", get_parameter_loader().get("saaaaaa.analysis.macro
_prompts.RoadmapOptimizer.__init__").get("auto_param_L1044_40", 0.5))
                std = peer.get("std", get_parameter_loader().get("saaaaaa.analysis.macro_p
rompts.RoadmapOptimizer.__init__").get("auto_param_L1045_38", 0.1))

                # Calculate z-score
                z = (score - mean) / std if std > 0 else get_parameter_loader().get("saaaa
aa.analysis.macro_prompts.RoadmapOptimizer.__init__").get("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

    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 = get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOp
timizer.__init__").get("auto_param_L1078_22", 0.1) * (len(low_performers) - self.k)
            adjusted *= (get_parameter_loader().get("saaaaaa.analysis.macro_prompts.Roadma
pOptimizer.__init__").get("auto_param_L1079_25", 1.0) - min(penalty, get_parameter_loader(
).get("saaaaaa.analysis.macro_prompts.RoadmapOptimizer.__init__").get("auto_param_L1079_44
", 0.5)))

        # Check if all within ±get_parameter_loader().get("saaaaaa.analysis.macro_prompts.
RoadmapOptimizer.__init__").get("auto_param_L1081_31", 0.5) z (tight distribution)
        all_tight = all(abs(z) <= get_parameter_loader().get("saaaaaa.analysis.macro_promp
ts.RoadmapOptimizer.__init__").get("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(get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOptim
izer.__init__").get("auto_param_L1088_19", 0.0), min(get_parameter_loader().get("saaaaaa.a
nalysis.macro_prompts.RoadmapOptimizer.__init__").get("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 > get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOptim
izer.__init__").get("auto_param_L1100_19", 0.5):
            return "above_average"
        elif avg_z < -get_parameter_loader().get("saaaaaa.analysis.macro_prompts.RoadmapOp
timizer.__init__").get("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", get_parameter_loader
().get("saaaaaa.analysis.macro_prompts.MacroPromptsOrchestrator.__init__").get("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", get_parameter_loader
().get("saaaaaa.analysis.macro_prompts.MacroPromptsOrchestrator.__init__").get("auto_param
_L1198_70", 1.0))
        )
        results["peer_normalization"] = asdict(peer_normalization)

        logger.info("Completed all 5 macro analyses")
        return results

===== FILE: src/saaaaaa/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 saaaaaa import get_parameter_loader
from saaaaaa.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: "Crítico",
    }[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 intraclúster muestra un CV de {cv:.2f} frente al referente de
{peer_cv:.2f}.",
        f"La brecha máxima es de {max_gap:.1f} puntos, lo que sitúa la clasificación en
nivel {classification}.",
        f"El coeficiente de Gini ({gini:.2f}) evidencia {'alta' if gini > 0.3 else
'moderada'} concentración de resultados.",
        "La penalización propuesta ψ 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 apreciación 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("saaaaaa.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", get_parameter_loader().get("saaaaaa.analysis.mes
o_cluster_analysis.MetricViolation.to_flag_dict").get("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(get_parameter_loader().get("saaaaaa.analysis.meso_cluster_
analysis.MetricViolation.to_flag_dict").get("auto_param_L306_32", 0.0), get_parameter_load
er().get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("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 = [get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.Metr
icViolation.to_flag_dict").get("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 = [get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.Metr
icViolation.to_flag_dict").get("auto_param_L334_19", 1.0)] * len(posts)

    # Prevent degenerate/negative totals; fallback to uniform if needed.
    weights = [max(get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.Metr
icViolation.to_flag_dict").get("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 ** get_parameter_loader().get("saaaaaa.analysis.mes
o_cluster_analysis.MetricViolation.to_flag_dict").get("auto_param_L345_42", 0.5))

    penalties_input = reconciliation_penalties or {}
    dispersion_penalty = float(penalties_input.get("dispersion_penalty", get_parameter_loa
der().get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto
_param_L348_73", 0.0)))
    coverage_penalty = float(penalties_input.get("coverage_penalty", get_parameter_loader(
).get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto_par
am_L349_69", 0.0)))
    reconciliation_penalty = float(penalties_input.get("reconciliation_penalty", get_param
eter_loader().get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").g
et("auto_param_L350_81", 0.0)))

    penalty_factor = reduce(
        lambda acc, val: acc * val,
        [
            max(get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.MetricV
iolation.to_flag_dict").get("auto_param_L355_16", 0.0), get_parameter_loader().get("saaaaa
a.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto_param_L355_21",
1.0) - dispersion_penalty),
            max(get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.MetricV
iolation.to_flag_dict").get("auto_param_L356_16", 0.0), get_parameter_loader().get("saaaaa
a.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto_param_L356_21",
1.0) - coverage_penalty),
            max(get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.MetricV
iolation.to_flag_dict").get("auto_param_L357_16", 0.0), get_parameter_loader().get("saaaaa
a.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto_param_L357_21",
1.0) - reconciliation_penalty),
        ],
        get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation
.to_flag_dict").get("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 dispersión, cobertura y reconciliación actúan 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 (σ ≈ {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 get_parameter_loader(
).get("saaaaaa.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict").get("auto_par
am_L395_69", 0.0)
    else:
        cluster_cv = get_parameter_loader().get("saaaaaa.analysis.meso_cluster_analysis.Me
tricViolation.to_flag_dict").get("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 desempeño heterogéneo por
área." ,
        f"Se ubican por encima del IQR {', '.join(above_areas) if above_areas else
'ninguna área'}, mientras que {', '.join(below_areas) if below_areas else 'no hay caídas
relevantes'} quedan por debajo.",
        f"Las áreas en zona intercuartílica ({', '.join(within_areas) if within_areas else
 'sin registros'}) sostienen la base del clúster.",
        "Los outliers detectados mediante Tukey advierten focos críticos que requieren
revisión específica.",
        f"Un municipio con media equiparable pero menor CV (~{cluster_cv:.2f}) ofrecería
narrativa más cohesionada, subrayando nuestra dispersión relativa.",
        "Conviene integrar estos hallazgos en la calibración narrativa para evitar
sobreponderar éxitos aislados frente a rezagos estructurales.",
        "Recomendar explicitar cómo la dispersión condiciona la comparabilidad con pares
que exhiben mayor equilibrio interno.",
    ]

    return json_payload, "\n".join(narrative_lines[:7])


===== FILE: src/saaaaaa/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 Question→Method 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
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from typing import Any
```

```
import numpy as np
from saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)

# ============================================================================
# PROVENANCE AUDITOR - QMCM INTEGRITY CHECK
# ============================================================================

@dataclass
class QMCMRecord:
    """Record in the Question→Method 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("saaaaaa.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("saaaaaa.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
    """

    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
        """
        self.p95_threshold = p95_latency_threshold or 100get_parameter_loader().get("saaaa
aa.analysis.micro_prompts.ProvenanceDAG.get_orphan_nodes").get("auto_param_L123_57", 0.0)
  # Default 1 second
        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("saaaaaa.analysis.micro_prompts.ProvenanceAuditor._check_latency_an
omalies")
    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("saaaaaa.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("saaaaaa.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 = get_parameter_loader().get("saaaaaa.analysis.micro_prompts.Pr
ovenanceAuditor.to_json").get("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 |Δ|
    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
    """

    def __init__(self, anti_miracle_cap: float = 0.95) -> None:
        """
        Initialize Bayesian posterior explainer

        Args:
            anti_miracle_cap: Maximum posterior probability (anti-miracle constraint)
        """
        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 |Δ|
        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 get_para
meter_loader().get("saaaaaa.analysis.micro_prompts.BayesianPosteriorExplainer.__init__").g
et("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("saaaaaa.analysis.micro_prompts.BayesianPosteriorExplainer._rank_si
gnals_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("saaaaaa.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("saaaaaa.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
(Δ={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 (Δ={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("saaaaaa.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("saaaaaa.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  # get_parameter_loader().get("saaaaaa.analysis.micro_prompts.CausalCh
ain.length").get("auto_param_L501_23", 0.0)-
get_parameter_loader().get("saaaaaa.analysis.micro_prompts.CausalChain.length").get("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
    """

    def __init__(self, fragility_threshold: float = 0.3) -> None:
        """
        Initialize stress tester

        Args:
            fragility_threshold: Support score drop threshold for fragility
        """
        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 > τ

        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 get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester.__init__").get("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 get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester.__init__").get("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 get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester.__init__").get("auto_param_L606_19", 1.0)  # Maximum drop if 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 get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester.__init__").get("auto_param_L624_19", 0.0)

        drop = (baseline_support - reduced_support) / baseline_support
        return max(get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester.__init__").get("auto_param_L627_19", 0.0), min(get_parameter_loader().get("saa
aaaa.analysis.micro_prompts.AntiMilagroStressTester.__init__").get("auto_param_L627_28",
1.0), drop))  # Clamp to [0, 1]

    @calibrated_method("saaaaaa.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 get_parameter_loader().get("saaaaaa.analysis.micro_prompts.AntiMilagroS
tressTester._calculate_support_score").get("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("saaaaaa.analysis.micro_prompts.AntiMilagroStressTester._generate_e
xplanation")
    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("saaaaaa.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 = get_parameter_loader().get("saaaa
aa.analysis.micro_prompts.AntiMilagroStressTester.to_json").get("auto_param_L668_57",
0.95)) -> BayesianPosteriorExplainer:
    """Factory function for BayesianPosteriorExplainer"""
    return BayesianPosteriorExplainer(anti_miracle_cap)

def create_stress_tester(fragility_threshold: float = get_parameter_loader().get("saaaaaa.
analysis.micro_prompts.AntiMilagroStressTester.to_json").get("auto_param_L672_54", 0.3))
-> AntiMilagroStressTester:
    """Factory function for AntiMilagroStressTester"""
    return AntiMilagroStressTester(fragility_threshold)

===== FILE: src/saaaaaa/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 saaaaaa import get_parameter_loader
from saaaaaa.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("saaaaaa.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("saaaaaa.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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._load_
canonical_notation")
    def _load_canonical_notation(self) -> None:
        """Load canonical notation for validation"""
        try:
            from saaaaaa.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("saaaaaa.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 .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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._load_
rules")
    def _load_rules(self) -> None:
        """Load and validate recommendation rules"""
        # Delegate to factory for I/O operation
        from .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("saaaaaa.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("saaaaaa.analysis.recommendation_engine.RecommendationEngine.get_th
resholds_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 (get_parameter_loader().get(
"saaaaaa.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith"
).get("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': get_parameter_loader().get("saaaaa
a.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith").get("
auto_param_L398_56", 0.15), 'weak_pa': 'PA02'},
                'CL02': {'score': 62.0, 'variance': get_parameter_loader().get("saaaaa
a.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith").get("
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 >= get_parameter_loader().get("saaaaaa.an
alysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith").get("auto
_param_L488_52", 0.08) or variance_level == 'MEDIA' and (variance < get_parameter_loader()
.get("saaaaaa.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_mono
lith").get("auto_param_L488_102", 0.08) or variance >= get_parameter_loader().get("saaaaaa
.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith").get("a
uto_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 < get_parameter_loader().get("saaaaaa.analysis.recommendat
ion_engine.RecommendationEngine.get_thresholds_from_monolith").get("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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._subst
itute_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._rende
r_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._valid
ate_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 .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("saaaaaa.analysis.recommendation_engine.RecommendationEngine._forma
t_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**Intervención:** {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')} →
{rec.horizon.get('end')}\n")
                lines.append("\n**Verificación:**")
                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/saaaaaa/analysis/report_assembly.py =====
"""
Report Assembly Module - Integrates with Questionnaire Monolith
================================================================

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 monolith hash

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 6: No reimplemented logic - delegates to provider

Author: Integration Team
Version: 1.0.0
Python: 3.10+
"""

from __future__ import annotations

import logging
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from typing import TYPE_CHECKING, Any
from saaaaaa.core.calibration.decorators import calibrated_method

if TYPE_CHECKING:
    from pathlib import Path

logger = logging.getLogger(__name__)


@dataclass
class ReportMetadata:
    """Metadata for analysis report with monolith traceability"""

    report_id: str
    generated_at: str
    monolith_version: str
    monolith_hash: str  # SHA-256 of questionnaire_monolith.json
    plan_name: str
    total_questions: int
    questions_analyzed: int
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass
class QuestionAnalysis:
    """Analysis result for a single micro question"""

    question_id: str
    question_global: int
    base_slot: str
    scoring_modality: str | None
    score: float | None
    evidence: list[str]
    patterns_applied: list[str]
    recommendation: str | None = None
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass
class AnalysisReport:
    """Complete policy analysis report"""

    metadata: ReportMetadata
    micro_analyses: list[QuestionAnalysis]
    meso_clusters: dict[str, Any]
    macro_summary: dict[str, Any]
    evidence_chain_hash: str | None = None

    @calibrated_method("saaaaaa.analysis.report_assembly.AnalysisReport.to_dict")
    def to_dict(self) -> dict[str, Any]:
        """Convert report to dictionary for JSON serialization"""
        return {
            'metadata': asdict(self.metadata),
            'micro_analyses': [asdict(q) for q in self.micro_analyses],
            'meso_clusters': self.meso_clusters,
            'macro_summary': self.macro_summary,
            'evidence_chain_hash': self.evidence_chain_hash
        }


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
    - Traceability via monolith hash
    """

    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.
        """
        self.questionnaire_provider = questionnaire_provider
        self.evidence_registry = evidence_registry
        self.qmcm_recorder = qmcm_recorder
        self.orchestrator = orchestrator

        logger.info("ReportAssembler initialized with dependency injection")

    def assemble_report(
        self,
        plan_name: str,
        execution_results: dict[str, Any],
        report_id: str | 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
        """
        # 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}"

        # Get questionnaire data and compute hash
        questionnaire_data = self.questionnaire_provider.get_data()

        # Import factory for hash computation (not for I/O)
        from ..core.orchestrator.factory import compute_monolith_hash
        monolith_hash = compute_monolith_hash(questionnaire_data)

        # Extract metadata
        version = questionnaire_data.get('version', 'unknown')
        blocks = questionnaire_data.get('blocks', {})
        micro_questions = blocks.get('micro_questions', [])

        # Create report metadata
        metadata = ReportMetadata(
            report_id=report_id,
            generated_at=datetime.now(timezone.utc).isoformat(),
            monolith_version=version,
            monolith_hash=monolith_hash,
            plan_name=plan_name,
            total_questions=len(micro_questions),
            questions_analyzed=len(execution_results.get('questions', {}))
        )

        # Assemble micro analyses
        micro_analyses = self._assemble_micro_analyses(
            micro_questions,
            execution_results
        )

        # 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

        report = AnalysisReport(
            metadata=metadata,
            micro_analyses=micro_analyses,
            meso_clusters=meso_clusters,
            macro_summary=macro_summary,
            evidence_chain_hash=evidence_chain_hash
        )

        logger.info(
            f"Report assembled: {report_id} "
            f"({len(micro_analyses)} questions, hash: {monolith_hash[:16]}...)"
        )

        return report

    def _assemble_micro_analyses(
        self,
        micro_questions: list[dict[str, Any]],
        execution_results: dict[str, Any]
    ) -> list[QuestionAnalysis]:
        """Assemble micro-level question analyses"""
        analyses = []
        question_results = execution_results.get('questions', {})

        for question in micro_questions:
            question_id = question.get('question_id', '')
            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
[]

            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'),
                score=result.get('score'),
                evidence=result.get('evidence', []),
                patterns_applied=pattern_names,
                recommendation=result.get('recommendation'),
                metadata={
                    'dimension': question.get('dimension'),
                    'policy_area': question.get('policy_area')
                }
            )
            analyses.append(analysis)

        return analyses

    def _assemble_meso_clusters(
        self,
        execution_results: dict[str, Any]
    ) -> dict[str, Any]:
        """Assemble meso-level cluster analyses"""
        return execution_results.get('meso_clusters', {})
    def _assemble_macro_summary(
        self,
        execution_results: dict[str, Any]
    ) -> dict[str, Any]:
        """Assemble macro-level summary"""
        return execution_results.get('macro_summary', {})

    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')

        NOTE: This delegates I/O to factory for architectural compliance.
        """
        # Delegate to factory for I/O
        from .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 ValueError(f"Unsupported format: {format}")

        logger.info(f"Report exported to {output_path} in {format} format")


@calibrated_method("saaaaaa.analysis.report_assembly.ReportAssembler._format_as_markdown")
    def _format_as_markdown(self, report: AnalysisReport) -> str:
        """Format report as Markdown"""
        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[:16]}...\n",
            f"**Questions Analyzed:**
{report.metadata.questions_analyzed}/{report.metadata.total_questions}\n",
            "\n## Micro-Level Analyses\n"
        ]

        for analysis in report.micro_analyses[:10]:  # Show first 10
            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) > 10:
            lines.append(f"\n_...and {len(report.micro_analyses) - 10} more questions_\n")

        lines.append("\n## Meso-Level Clusters\n")
        lines.append(f"```json\n{report.meso_clusters}\n```\n")

        lines.append("\n## Macro Summary\n")
        lines.append(f"```json\n{report.macro_summary}\n```\n")

        if report.evidence_chain_hash:
            lines.append(f"\n**Evidence Chain Hash:** {report.evidence_chain_hash}\n")
```

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


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
    """
    return ReportAssembler(
        questionnaire_provider=questionnaire_provider,
        evidence_registry=evidence_registry,
        qmcm_recorder=qmcm_recorder,
        orchestrator=orchestrator
    )


__all__ = [
    'ReportMetadata',
    'QuestionAnalysis',
    'AnalysisReport',
    'ReportAssembler',
    'create_report_assembler',
]

===== FILE: src/saaaaaa/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: saaaaaa/analysis/scoring.py
Import them directly from there: `from saaaaaa.analysis.scoring import Evidence`
"""

# Import from this package's scoring.py
from .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 saaaaaa/analysis/scoring.py (module)
# This __init__.py is for saaaaaa/analysis/scoring/ (package)
# These are SEPARATE namespaces. Import Evidence from the module directly.

===== FILE: src/saaaaaa/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 saaaaaa import get_parameter_loader
from saaaaaa.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("saaaaaa.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("saaaaaa.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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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=(get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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"✓ Evidence validation passed for {modality.value}")
        except (EvidenceStructureError, ModalityValidationError) as e:
            logger.exception(f"✗ 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 get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.ModalityConfig
.validate_evidence").get("auto_param_L335_15", 0.0) <= numeric_value <= get_parameter_load
er().get("saaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence").get("auto_pa
ram_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", get_parameter_loader().get("saaaaaa.analysis.s
coring.scoring.ModalityConfig.validate_evidence").get("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 get_parameter_loader().ge
t("saaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence").get("auto_param_L38
5_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", get_parameter_loader().get("saaaaaa.analys
is.scoring.scoring.ModalityConfig.validate_evidence").get("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", get_parameter_loader().get("saaaaaa.
analysis.scoring.scoring.ModalityConfig.validate_evidence").get("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 = get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.
ModalityConfig.validate_evidence").get("auto_param_L597_29", 1.0) if traceability else get
_parameter_loader().get("saaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence
").get("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 get_parameter_loader().get("
saaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence").get("auto_param_L610_6
2", 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", get_parameter_loader().get("saaaaaa.analys
is.scoring.scoring.ModalityConfig.validate_evidence").get("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 get_parameter_loader().get("s
aaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence").get("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
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: >= get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Modal
ityConfig.validate_evidence").get("auto_param_L706_24", 0.85)
        - BUENO: >= get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.ModalityC
onfig.validate_evidence").get("auto_param_L707_20", 0.70)
        - ACEPTABLE: >= get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Modal
ityConfig.validate_evidence").get("auto_param_L708_24", 0.55)
        - INSUFICIENTE: < get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Mod
alityConfig.validate_evidence").get("auto_param_L709_26", 0.55)
    """
    if thresholds is None:
        thresholds = {
            "EXCELENTE": get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("auto_param_L713_25", 0.85),
            "BUENO": get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Modality
Config.validate_evidence").get("auto_param_L714_21", 0.70),
            "ACEPTABLE": get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.Moda
lityConfig.validate_evidence").get("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), get_parameter_loader().get("saaaaaa.
analysis.scoring.scoring.ModalityConfig.validate_evidence").get("auto_param_L721_54",
0.0), get_parameter_loader().get("saaaaaa.analysis.scoring.scoring.ModalityConfig.validate
_evidence").get("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, get_parameter_loader().get("saaaaaa.analysi
s.scoring.scoring.ModalityConfig.validate_evidence").get("auto_param_L826_47", 0.0), get_p
arameter_loader().get("saaaaaa.analysis.scoring.scoring.ModalityConfig.validate_evidence")
.get("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"✓ 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/saaaaaa/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: ≥ 0.85 (85th percentile) - green indicator
- GOOD: ≥ 0.70 (70th percentile) - blue indicator
- ACCEPTABLE: ≥ 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 saaaaaa import get_parameter_loader
from saaaaaa.core.calibration.decorators import calibrated_method

logger = logging.getLogger(__name__)
```

```
# ============================================================================
# ENUMS - EXACTOS DEL MONOLITH
# ============================================================================

class ScoringModality(Enum):
    """Modalidades de scoring del monolith (línea 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 (línea 34513)."""
    EXCELENTE = "EXCELENTE"    # ≥ 0.85
    BUENO = "BUENO"            # ≥ 0.70
    ACEPTABLE = "ACEPTABLE"   # ≥ 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 extraída 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 módulo.
    """
    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 según 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 configuración del monolith.

        Args:
            config: Configuración de scoring (defaults del monolith si None)
        """
        self.config = config or ScoringConfig()
        self.logger = logger

    # ========================================================================
    # MÉTODO 1: SCORE TYPE_A
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_a")
    def score_type_a(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÉTODO 1: TYPE_A - Count 4 elements and scale to 0-3.

        ESPECIFICACIÓN (línea 34568 del monolith):
        - Aggregation: "presence_threshold"
        - Threshold: get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionSco
rer.score_type_a").get("auto_param_L245_21", 0.7)
        - Max_score: 3
        - Expected_elements: 4

        LÓGICA:
        1. Contar elementos encontrados (expected: 4)
        2. Calcular ratio = found / 4
        3. Si ratio >= get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionS
corer.score_type_a").get("auto_param_L252_23", 0.7): aplicar escala proporcional
        4. Si ratio < get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionSc
orer.score_type_a").get("auto_param_L253_22", 0.7): penalizar fuertemente

        ESCALA:
        - 4/4 elementos (100%) → 3.0
        - 3/4 elementos (75%) → 2.25
        - 2/4 elementos (50%) → penalizado
        - 1/4 elementos (25%) → penalizado
        - 0/4 elementos (0%) → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQ
uestionScorer.score_type_a").get("auto_param_L260_31", 0.0)

        Args:
            evidence: Evidencia extraída 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 get_parameter_loader().get(
"saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_a").get("auto_param_L274_63",
0.0)

        # Aplicar threshold del monolith
        if ratio >= threshold:
            # Escala proporcional: ratio * max_score
            score = ratio * max_score
        else:
            # Penalización: escala cuadrática para ratios bajos
            score = (ratio / threshold) * (ratio * max_score)

        # Clip al rango [0, max_score]
        score = max(get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScor
er.score_type_a").get("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}) →
score={score:.2f}")

        return score, details

    # ========================================================================
    # MÉTODO 2: SCORE TYPE_B
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_b")
    def score_type_b(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÉTODO 2: TYPE_B - Count up to 3 elements, each worth 1 point.

        ESPECIFICACIÓN (línea 34574 del monolith):
        - Aggregation: "binary_sum"
        - Max_score: 3
        - Max_elements: 3

        LÓGICA:
        1. Contar elementos encontrados (max: 3)
        2. Cada elemento = 1 punto
        3. Score = min(elements_found, 3)

        ESCALA:
        - 3+ elementos → 3.0
        - 2 elementos → 2.0
        - 1 elemento → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionS
corer.score_type_b").get("auto_param_L324_23", 1.0)
        - 0 elementos → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestion
Scorer.score_type_b").get("auto_param_L325_24", 0.0)

        Args:
            evidence: Evidencia extraída 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 → score={score:.2f}")

        return score, details

    # ========================================================================
    # MÉTODO 3: SCORE TYPE_C
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_c")
    def score_type_c(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÉTODO 3: TYPE_C - Count 2 elements and scale to 0-3.

        ESPECIFICACIÓN (línea 34580 del monolith):
        - Aggregation: "presence_threshold"
        - Threshold: get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionSco
rer.score_type_c").get("auto_param_L366_21", 0.5)
        - Max_score: 3
        - Expected_elements: 2

        LÓGICA:
        1. Contar elementos encontrados (expected: 2)
        2. Calcular ratio = found / 2
        3. Si ratio >= get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionS
corer.score_type_c").get("auto_param_L373_23", 0.5): aplicar escala proporcional
        4. Si ratio < get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionSc
orer.score_type_c").get("auto_param_L374_22", 0.5): penalizar

        ESCALA:
        - 2/2 elementos (100%) → 3.0
        - 1/2 elementos (50%) → 1.5
        - 0/2 elementos (0%) → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQ
uestionScorer.score_type_c").get("auto_param_L379_31", 0.0)

        Args:
            evidence: Evidencia extraída 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 get_parameter_loader().get(
"saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_c").get("auto_param_L393_63",
0.0)

        # Aplicar threshold del monolith
        if ratio >= threshold:
            # Escala proporcional
            score = ratio * max_score
        else:
            # Penalización cuadrática
            score = (ratio / threshold) * (ratio * max_score)

        # Clip al rango [0, max_score]
        score = max(get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScor
er.score_type_c").get("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}) →
score={score:.2f}")

        return score, details

    # ========================================================================
    # MÉTODO 4: SCORE TYPE_D
    # ========================================================================

    @calibrated_method("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_d")
    def score_type_d(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÉTODO 4: TYPE_D - Count 3 elements, weighted [get_parameter_loader().get("saaaaaa
.analysis.scoring.MicroQuestionScorer.score_type_d").get("auto_param_L428_55", 0.4), get_p
arameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_d").get("au
to_param_L428_60", 0.3), get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestio
nScorer.score_type_d").get("auto_param_L428_65", 0.3)].

        ESPECIFICACIÓN (línea 34586 del monolith):
        - Aggregation: "weighted_sum"
        - Weights: [get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScor
er.score_type_d").get("auto_param_L432_20", 0.4), get_parameter_loader().get("saaaaaa.anal
ysis.scoring.MicroQuestionScorer.score_type_d").get("auto_param_L432_25", 0.3), get_parame
ter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_d").get("auto_pa
ram_L432_30", 0.3)]
        - Max_score: 3
        - Expected_elements: 3

        LÓGICA:
        1. Se esperan 3 elementos con importancia diferente
        2. Elemento 1: peso get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQues
tionScorer.score_type_d").get("auto_param_L438_28", 0.4) (más importante)
        3. Elemento 2: peso get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQues
tionScorer.score_type_d").get("auto_param_L439_28", 0.3)
        4. Elemento 3: peso get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQues
tionScorer.score_type_d").get("auto_param_L440_28", 0.3)
        5. Score = (sum of weights for found elements) * max_score

        ESCALA:
        - 3 elementos (todos) → weights_sum=get_parameter_loader().get("saaaaaa.analysis.s
coring.MicroQuestionScorer.score_type_d").get("auto_param_L444_44", 1.0) → 3.0
        - 2 elementos (ej: elem1+elem2) → weights_sum=get_parameter_loader().get("saaaaaa.
analysis.scoring.MicroQuestionScorer.score_type_d").get("auto_param_L445_54", 0.7) → 2.1
        - 1 elemento (ej: elem1) → weights_sum=get_parameter_loader().get("saaaaaa.analysi
s.scoring.MicroQuestionScorer.score_type_d").get("auto_param_L446_47", 0.4) → 1.2
        - 0 elementos → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestion
Scorer.score_type_d").get("auto_param_L447_24", 0.0)

        Args:
            evidence: Evidencia extraída 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 está ordenado por importancia
        # o usamos confidence_scores si están 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(get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestionScor
er.score_type_d").get("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} → score={score:.2f}")

        return score, details

    # ==========================================================================
    # MÉTODO 5: SCORE TYPE_E
    # ==========================================================================

    @calibrated_method("saaaaaa.analysis.scoring.MicroQuestionScorer.score_type_e")
    def score_type_e(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
        """
        MÉTODO 5: TYPE_E - Boolean presence check.

        ESPECIFICACIÓN (línea 34596 del monolith):
        - Aggregation: "binary_presence"
        - Max_score: 3
        LÓGICA:
        1. Verificar si existe evidencia (binario: sí/no)
        2. Si existe: 3.0
        3. Si no existe: get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQuestio
nScorer.score_type_e").get("auto_param_L511_25", 0.0)

        ESCALA:
        - Evidencia presente → 3.0
        - Evidencia ausente → get_parameter_loader().get("saaaaaa.analysis.scoring.MicroQu
estionScorer.score_type_e").get("auto_param_L515_30", 0.0)

        Args:
            evidence: Evidencia extraída

        Returns:
            Tuple de (score, details)
        """
        max_score = self.config.type_e_max_score
```