

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 6
3: =====
4: Generated: 2025-12-07T06:17:16.619183
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfan_pipeline/analysis/contradiction_deteccion.py
11: =====
12:
13: """
14: Advanced Policy Contradiction Detection System for Colombian Municipal Development Plans
15:
16: Este sistema implementa el estado del arte en detección de contradicciones para análisis
17: de políticas públicas, específicamente calibrado para Planes de Desarrollo Municipal (PDM)
18: colombianos según la Ley 152 de 1994 y metodología DNP.
19:
20: Innovations:
21: - Transformer-based semantic similarity using sentence-transformers
22: - Graph-based contradiction reasoning with NetworkX
23: - Bayesian inference for confidence scoring
24: - Temporal logic verification for timeline consistency
25: - Multi-dimensional vector embeddings for policy alignment
26: - Statistical hypothesis testing for numerical claims
27: """
28:
29: from __future__ import annotations
30:
31: import logging
32: import re
33: from dataclasses import dataclass, field
34: from enum import Enum, auto
35: from typing import Any
36:
37: import networkx as nx
38: import numpy as np
39: import torch
40: from scipy import stats
41: from scipy.spatial.distance import cosine
42: from scipy.stats import beta
43: from sentence_transformers import SentenceTransformer
44: from sklearn.feature_extraction.text import TfidfVectorizer
45: from sklearn.metrics.pairwise import cosine_similarity
46: from transformers import AutoModelForSequenceClassification, DebertaV2Tokenizer, pipeline
47:
48: # Check dependency lockdown
49: from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
50:
51: # Import runtime error fixes for defensive programming
52: from farfan_pipeline.utils.runtime_error_fixes import ensure_list_return, safe_text_extract
53: from farfan_pipeline.core.parameters import ParameterLoaderV2
54: from farfan_pipeline.core.calibration.decorators import calibrated_method
55:
56: _lockdown = get_dependency_lockdown()
```

```
57:
58: # Configure logging with structured format
59: logging.basicConfig(
60:     level=logging.INFO,
61:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
62: )
63: logger = logging.getLogger(__name__)
64:
65: class ContradictionType(Enum):
66:     """TaxonomÃ-a de contradicciones segÃ³n estÃ;ndares de polÃ-tica pÃºblica"""
67:     NUMERICAL_INCONSISTENCY = auto()
68:     TEMPORAL_CONFLICT = auto()
69:     SEMANTIC_OPPOSITION = auto()
70:     LOGICAL_INCOMPATIBILITY = auto()
71:     RESOURCE_ALLOCATION_MISMATCH = auto()
72:     OBJECTIVE_MISALIGNMENT = auto()
73:     REGULATORY_CONFLICT = auto()
74:     STAKEHOLDER_DIVERGENCE = auto()
75:
76: class PolicyDimension(Enum):
77:     """Dimensiones del Plan de Desarrollo segÃ³n DNP Colombia"""
78:     DIAGNOSTICO = "diagnÃ³stico"
79:     ESTRATEGICO = "estratÃ©gico"
80:     PROGRAMATICO = "programÃ;tico"
81:     FINANCIERO = "plan plurianual de inversiones"
82:     SEGUIMIENTO = "seguimiento y evaluaciÃ³n"
83:     TERRITORIAL = "ordenamiento territorial"
84:
85: @dataclass(frozen=True)
86: class PolicyStatement:
87:     """RepresentaciÃ³n estructurada de una declaraciÃ³n de polÃ-tica"""
88:     text: str
89:     dimension: PolicyDimension
90:     position: tuple[int, int] # (start, end) in document
91:     entities: list[str] = field(default_factory=list)
92:     temporal_markers: list[str] = field(default_factory=list)
93:     quantitative_claims: list[dict[str, Any]] = field(default_factory=list)
94:     embedding: np.ndarray | None = None
95:     context_window: str = ""
96:     semantic_role: str | None = None
97:     dependencies: set[str] = field(default_factory=set)
98:
99: @dataclass
100: class ContradictionEvidence:
101:     """Evidencia estructurada de contradicciÃ³n con trazabilidad completa"""
102:     statement_a: PolicyStatement
103:     statement_b: PolicyStatement
104:     contradiction_type: ContradictionType
105:     confidence: float # Bayesian posterior probability
106:     severity: float # Impact on policy coherence
107:     semantic_similarity: float
108:     logical_conflict_score: float
109:     temporal_consistency: bool
110:     numerical_divergence: float | None
111:     affected_dimensions: list[PolicyDimension]
112:     resolution_suggestions: list[str]
```

```
113:     graph_path: list[str] | None = None
114:     statistical_significance: float | None = None
115:
116: class BayesianConfidenceCalculator:
117:     """
118:         Bayesian confidence calculator with domain-informed priors.
119:
120:     Uses Beta distribution priors calibrated from empirical analysis of
121:     Colombian municipal development plans (PDMs).
122:     """
123:
124:     def __init__(self) -> None:
125:         # Priors based on empirical analysis of Colombian municipal development plans (PDMs)
126:         self.prior_alpha = 2.5 # Shape parameter for beta distribution
127:         self.prior_beta = 7.5 # Scale parameter (conservative bias favoring lower confidence)
128:
129:     def calculate_posterior(
130:         self,
131:         evidence_strength: float,
132:         observations: int,
133:         domain_weight: float = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L
121_35", 1.0)
134:     ) -> float:
135:         """
136:             Calculate posterior probability using Bayesian inference.
137:
138:             Updates the Beta distribution prior with observed evidence to compute
139:             the posterior mean, which represents the confidence level in the finding.
140:
141:             Args:
142:                 evidence_strength: Strength of the evidence (ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.
121_35", "auto_param_L130_57", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L130_61
121_35", 1.0) scale, unitless ratio)
143:                     observations: Number of observations supporting the evidence (count)
144:                     domain_weight: Policy domain-specific weight (multiplier, default: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianC
onfidenceCalculator.__init__", "auto_param_L132_79", 1.0))
145:
146:             Returns:
147:                 float: Posterior probability (ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_
param_L135_42", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L135_46", 1.0) scale)
representing confidence level
148:         """
149:         # Update Beta distribution with evidence
150:         alpha_post = self.prior_alpha + evidence_strength * observations * domain_weight
151:         beta_post = self.prior_beta + (1 - evidence_strength) * observations * domain_weight
152:
153:         # Calculate mean of posterior distribution
154:         posterior_mean = alpha_post / (alpha_post + beta_post)
155:
156:         # Calculate 95% credible interval
157:         credible_interval = beta.interval(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_
param_L145_42", 0.95), alpha_post, beta_post)
158:
159:         # Adjust for uncertainty (wider intervals reduce confidence)
160:         uncertainty_penalty = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L148_3
0", 1.0) - (credible_interval[1] - credible_interval[0])
```

```
161:
162:         return min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.BayesianConfidenceCalculator.__init__", "auto_param_L150_19", 1.0), p
163: posterior_mean * uncertainty_penalty)
164: class TemporalLogicVerifier:
165:     """
166:         Temporal consistency verification using Linear Temporal Logic (LTL).
167:
168:         Analyzes policy statements for temporal contradictions, deadline violations,
169:         and ordering conflicts using temporal logic patterns.
170:     """
171:
172:     def __init__(self) -> None:
173:         self.temporal_patterns = {
174:             'sequential': re.compile(r'(primero|luego|despuÃ±os|posteriormente|finalmente)', re.IGNORECASE),
175:             'parallel': re.compile(r'(simultÃ;neamente|al mismo tiempo|paralelamente)', re.IGNORECASE),
176:             'deadline': re.compile(r'(antes de|hasta|mÃ;ximo|plazo)', re.IGNORECASE),
177:             'milestone': re.compile(r'(hito|meta intermedia|checkpoint)', re.IGNORECASE)
178:         }
179:
180:     def verify_temporal_consistency(
181:         self,
182:         statements: list[PolicyStatement]
183:     ) -> tuple[bool, list[dict[str, Any]]]:
184:         """
185:             Verify temporal consistency between policy statements.
186:
187:             Analyzes temporal ordering and deadline constraints to identify
188:             contradictions or violations in the policy timeline.
189:
190:             Args:
191:                 statements: List of policy statements to analyze
192:
193:             Returns:
194:                 tuple[bool, list[dict]]: A tuple containing:
195:                     - is_consistent: True if no conflicts found
196:                     - conflicts_found: List of detected temporal conflicts
197:         """
198:         timeline = self._build_timeline(statements)
199:         conflicts = []
200:
201:         # Verify temporal ordering
202:         for i, event_a in enumerate(timeline):
203:             for event_b in timeline[i + 1:]:
204:                 if self._has_temporal_conflict(event_a, event_b):
205:                     conflicts.append({
206:                         'event_a': event_a,
207:                         'event_b': event_b,
208:                         'conflict_type': 'temporal_ordering'
209:                     })
210:
211:         # Verify deadline constraints
212:         deadline_violations = self._check_deadline_constraints(timeline)
213:         conflicts.extend(deadline_violations)
214:
215:         return len(conflicts) == 0, conflicts
```

```
216:
217:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._build_timeline")
218:     def _build_timeline(self, statements: list[PolicyStatement]) -> list[dict]:
219:         """
220:             Build timeline from policy statements.
221:
222:             Extracts temporal markers and organizes them chronologically.
223:
224:             Args:
225:                 statements: List of policy statements
226:
227:             Returns:
228:                 list[dict]: Sorted timeline events with timestamps
229:             """
230:             timeline = []
231:             for stmt in statements:
232:                 for marker in stmt.temporal_markers:
233:                     # Extract structured temporal information
234:                     timeline.append({
235:                         'statement': stmt,
236:                         'marker': marker,
237:                         'timestamp': self._parse_temporal_marker(marker),
238:                         'type': self._classify_temporal_type(marker)
239:                     })
240:             return sorted(timeline, key=lambda x: x.get('timestamp', 0))
241:
242:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._parse_temporal_marker")
243:     def _parse_temporal_marker(self, marker: str) -> int | None:
244:         """
245:             Parse temporal marker to numeric timestamp.
246:
247:             Implements Colombian policy document temporal format parsing.
248:
249:             Args:
250:                 marker: Temporal marker string (e.g., "2024", "Q2", "segundo trimestre")
251:
252:             Returns:
253:                 int | None: Numeric timestamp, or None if parsing fails
254:             """
255:             # Implementation specific to Colombian policy document format
256:             year_match = re.search(r'20\d{2}', marker)
257:             if year_match:
258:                 return int(year_match.group())
259:
260:             quarter_patterns = {
261:                 'primer': 1, 'segundo': 2, 'tercer': 3, 'cuarto': 4,
262:                 'Q1': 1, 'Q2': 2, 'Q3': 3, 'Q4': 4
263:             }
264:             for pattern, quarter in quarter_patterns.items():
265:                 if pattern in marker.lower():
266:                     return quarter
267:
268:             return None
269:
270:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._has_temporal_conflict")
271:     def _has_temporal_conflict(self, event_a: dict, event_b: dict) -> bool:
```



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

333:
334:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.TemporalLogicVerifier._classify_temporal_type")
335:     def _classify_temporal_type(self, marker: str) -> str:
336:         """Clasifica el tipo de marcador temporal"""
337:         for pattern_type, pattern in self.temporal_patterns.items():
338:             if pattern.search(marker):
339:                 return pattern_type
340:         return 'unspecified'  

341:
342: class PolicyContradictionDetector:
343:     """
344:         Sistema avanzado de detección de contradicciones para PDMs colombianos.
345:         Implementa el estado del arte en NLP y razonamiento lógico.
346:     """
347:
348:     def __init__(
349:         self,
350:         model_name: str = "hiiamsid/sentence_similarity_spanish_es",
351:         spacy_model: str = "es_core_news_lg",
352:         device: str = "cuda" if torch.cuda.is_available() else "cpu"
353:     ) -> None:
354:         # Modelos de transformers para análisis semántico
355:         self.semantic_model = SentenceTransformer(model_name, device=device)
356:
357:         # Modelo de clasificación de contradicciones
358:         model_name = "microsoft/deberta-v3-base"
359:         tokenizer = DebertaV2Tokenizer.from_pretrained(model_name)
360:         model = AutoModelForSequenceClassification.from_pretrained(model_name)
361:
362:         self.contradiction_classifier = pipeline(
363:             "text-classification",
364:             model=model,
365:             tokenizer=tokenizer,
366:             device=0 if device == "cuda" else -1,
367:         )
368:
369:         # Procesamiento de lenguaje natural
370:         # Delegate to factory for I/O operation
371:         from farfan_pipeline.analysis.factory import load_spacy_model
372:         self.nlp = load_spacy_model(spacy_model)
373:
374:         # Componentes especializados
375:         self.bayesian_calculator = BayesianConfidenceCalculator()
376:         self.temporal_verifier = TemporalLogicVerifier()
377:
378:         # Grafo de conocimiento para razonamiento
379:         self.knowledge_graph = nx.DiGraph()
380:
381:         # Vectorizador TF-IDF para análisis complementario
382:         self.tfidf = TfidfVectorizer(
383:             ngram_range=(1, 3),
```

```
384:             max_features=5000,
385:             sublinear_tf=True
386:         )
387:
388:         # Patrones especÃ–ficos de PDM colombiano
389:         self._initialize_pdm_patterns()
390:
391:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._initialize_pdm_patterns")
392:     def _initialize_pdm_patterns(self) -> None:
393:         """Inicializa patrones especÃ–ficos de PDMs colombianos"""
394:         self.pdm_patterns = {
395:             'ejes_estrategicos': re.compile(
396:                 r'(eje\stestratÃ©gico|lÃ–nea\s+estratÃ©gica|pilar|dimensiÃ³n)',
397:                 re.IGNORECASE
398:             ),
399:             'programas': re.compile(
400:                 r'(programa|subprograma|proyecto|iniciativa)',
401:                 re.IGNORECASE
402:             ),
403:             'metas': re.compile(
404:                 r'(meta\s+de\s+resultado|meta\s+de\s+producto|indicador)',
405:                 re.IGNORECASE
406:             ),
407:             'recursos': re.compile(
408:                 r'(SGP|regalÃ–as|recursos\s+propios|cofinaciÃ³n|crÃ©dito)',
409:                 re.IGNORECASE
410:             ),
411:             'normativa': re.compile(
412:                 r'(ley\s+\d+|decreto\s+\d+|acuerdo\s+\d+|resoluciÃ³n\s+\d+)',
413:                 re.IGNORECASE
414:             )
415:         }
416:
417:     def detect(
418:         self,
419:         text: str,
420:         plan_name: str = "PDM",
421:         dimension: PolicyDimension = PolicyDimension.ESTRATEGICO
422:     ) -> dict[str, Any]:
423:         """
424:             Detecta contradicciones con anÃ¡lisis multi-dimensional avanzado
425:
426:             Args:
427:                 text: Texto del plan de desarrollo
428:                 plan_name: Nombre del PDM
429:                 dimension: DimensiÃ³n del plan siendo analizada
430:
431:             Returns:
432:                 AnÃ¡lisis completo con contradicciones detectadas y mÃ©tricas
433:         """
434:         # Extraer declaraciones de polÃ–tica estructuradas
435:         statements = self._extract_policy_statements(text, dimension)
436:
437:         # Generar embeddings semÃ–nticos
438:         statements = self._generate_embeddings(statements)
439:
```

```
440:     # Construir grafo de conocimiento
441:     self._build_knowledge_graph(statements)
442:
443:     # Detectar contradicciones multi-tipo
444:     contradictions = []
445:
446:     # 1. Contradicciones semánticas usando transformers
447:     semantic_contradictions = self._detect_semantic_contradictions(statements)
448:     contradictions.extend(ensure_list_return(semantic_contradictions))
449:
450:     # 2. Inconsistencias numéricas con pruebas estadísticas
451:     numerical_contradictions = self._detect_numerical_inconsistencies(statements)
452:     contradictions.extend(ensure_list_return(numerical_contradictions))
453:
454:     # 3. Conflictos temporales con verificación lógica
455:     temporal_conflicts = self._detect_temporal_conflicts(statements)
456:     contradictions.extend(ensure_list_return(temporal_conflicts))
457:
458:     # 4. Incompatibilidades lógicas usando razonamiento en grafo
459:     logical_contradictions = self._detect_logical_incompatibilities(statements)
460:     contradictions.extend(ensure_list_return(logical_contradictions))
461:
462:     # 5. Conflictos de asignación de recursos
463:     resource_conflicts = self._detect_resource_conflicts(statements)
464:     contradictions.extend(ensure_list_return(resource_conflicts))
465:
466:     # Calcular métricas agregadas
467:     coherence_metrics = self._calculate_coherence_metrics(
468:         contradictions,
469:         statements,
470:         text
471:     )
472:
473:     # Generar recomendaciones de resolución
474:     recommendations = self._generate_resolution_recommendations(contradictions)
475:
476:     return {
477:         "plan_name": plan_name,
478:         "dimension": dimension.value,
479:         "contradictions": [self._serialize_contradiction(c) for c in contradictions],
480:         "total_contradictions": len(contradictions),
481:         "high_severity_count": sum(1 for c in contradictions if c.severity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyC
ontradictionDetector._initialize_pdm_patterns", "auto_param_L469_81", 0.7)),
482:         "coherence_metrics": coherence_metrics,
483:         "recommendations": recommendations,
484:         "knowledge_graph_stats": self._get_graph_statistics()
485:     }
486:
487:     def _extract_policy_statements(
488:         self,
489:         text: str,
490:         dimension: PolicyDimension
491:     ) -> list[PolicyStatement]:
492:         """Extrae declaraciones de política estructuradas del texto"""
493:         doc = self.nlp(text)
494:         statements = []
```

```
495:
496:     for sent in doc.sents:
497:         # Analizar entidades nombradas
498:         entities = [ent.text for ent in sent.ents]
499:
500:         # Extraer marcadores temporales
501:         temporal_markers = self._extract_temporal_markers(sent.text)
502:
503:         # Extraer afirmaciones cuantitativas
504:         quantitative_claims = self._extract_quantitative_claims(sent.text)
505:
506:         # Determinar rol semántico
507:         semantic_role = self._determine_semantic_role(sent)
508:
509:         # Identificar dependencias
510:         dependencies = self._identify_dependencies(sent, doc)
511:
512:         statement = PolicyStatement(
513:             text=sent.text,
514:             dimension=dimension,
515:             position=(sent.start_char, sent.end_char),
516:             entities=entities,
517:             temporal_markers=temporal_markers,
518:             quantitative_claims=quantitative_claims,
519:             context_window=self._get_context_window(text, sent.start_char, sent.end_char),
520:             semantic_role=semantic_role,
521:             dependencies=dependencies
522:         )
523:
524:         statements.append(statement)
525:
526:     return statements
527:
528: def _generate_embeddings(
529:     self,
530:     statements: list[PolicyStatement]
531: ) -> list[PolicyStatement]:
532:     """Genera embeddings semánticos para las declaraciones"""
533:     texts = [stmt.text for stmt in statements]
534:     embeddings = self.semantic_model.encode(texts, convert_to_numpy=True)
535:
536:     # Crear nuevas instancias con embeddings
537:     enhanced_statements = []
538:     for stmt, embedding in zip(statements, embeddings, strict=False):
539:         enhanced = PolicyStatement(
540:             text=stmt.text,
541:             dimension=stmt.dimension,
542:             position=stmt.position,
543:             entities=stmt.entities,
544:             temporal_markers=stmt.temporal_markers,
545:             quantitative_claims=stmt.quantitative_claims,
546:             embedding=embedding,
547:             context_window=stmt.context_window,
548:             semantic_role=stmt.semantic_role,
549:             dependencies=stmt.dependencies
550:         )
551:
```

```

551:         enhanced_statements.append(enhanced)
552:
553:     return enhanced_statements
554:
555:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph")
556:     def _build_knowledge_graph(self, statements: list[PolicyStatement]) -> None:
557:         """Construye grafo de conocimiento para razonamiento"""
558:         self.knowledge_graph.clear()
559:
560:         for i, stmt in enumerate(statements):
561:             node_id = f"stmt_{i}"
562:             self.knowledge_graph.add_node(
563:                 node_id,
564:                 text=stmt.text[:100],
565:                 dimension=stmt.dimension.value,
566:                 entities=stmt.entities,
567:                 semantic_role=stmt.semantic_role
568:             )
569:
570:             # Conectar con declaraciones relacionadas
571:             for j, other in enumerate(statements):
572:                 if i != j:
573:                     similarity = self._calculate_similarity(stmt, other)
574:                     if similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph",
575: "auto_param_L562_36", 0.7): # Umbral de relaciÃ³n
576:                         self.knowledge_graph.add_edge(
577:                             f"stmt_{i}",
578:                             f"stmt_{j}",
579:                             weight=similarity,
580:                             relation_type=self._determine_relation_type(stmt, other)
581:                         )
582:
583:     def _detect_semantic_contradictions(
584:         self,
585:         statements: list[PolicyStatement]
586:     ) -> list[ContradictionEvidence]:
587:         """Detecta contradicciones semÃ¡nticas usando transformers"""
588:         contradictions = []
589:
590:         for i, stmt_a in enumerate(statements):
591:             for stmt_b in statements[i + 1:]:
592:                 if stmt_a.embedding is not None and stmt_b.embedding is not None:
593:                     # Calcular similaridad coseno
594:                     similarity = 1 - cosine(stmt_a.embedding, stmt_b.embedding)
595:
596:                     # Verificar contradicciÃ³n usando clasificador
597:                     combined_text = f"{stmt_a.text} [SEP] {stmt_b.text}"
598:                     contradiction_score = self._classify_contradiction(combined_text)
599:
600:                     if contradiction_score > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledg
e_graph", "auto_param_L587_45", 0.7) and similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowle
dge_graph", "auto_param_L587_66", 0.5):
601:                         # Calcular confianza Bayesiana
602:                         confidence = self.bayesian_calculator.calculate_posterior(
603:                             evidence_strength=contradiction_score,
604:                             observations=len(stmt_a.entities) + len(stmt_b.entities),

```

```

604:                 domain_weight=self._get_domain_weight(stmt_a.dimension)
605:             )
606:
607:             evidence = ContradictionEvidence(
608:                 statement_a=stmt_a,
609:                 statement_b=stmt_b,
610:                 contradiction_type=ContradictionType.SEMANTIC_OPPOSITION,
611:                 confidence=confidence,
612:                 severity=self._calculate_severity(stmt_a, stmt_b),
613:                 semantic_similarity=similarity,
614:                 logical_conflict_score=contradiction_score,
615:                 temporal_consistency=True,
616:                 numerical_divergence=None,
617:                 affected_dimensions=[stmt_a.dimension, stmt_b.dimension],
618:                 resolution_suggestions=self._suggest_resolutions(
619:                     ContradictionType.SEMANTIC_OPPOSITION
620:                 )
621:             )
622:             contradictions.append(evidence)
623:
624:     return contradictions
625:
626: def _detect_numerical_inconsistencies(
627:     self,
628:     statements: list[PolicyStatement]
629: ) -> list[ContradictionEvidence]:
630:     """Detecta inconsistencias numéricas con análisis estadístico"""
631:     contradictions = []
632:
633:     for i, stmt_a in enumerate(statements):
634:         for stmt_b in statements[i + 1:]:
635:             if stmt_a.quantitative_claims and stmt_b.quantitative_claims:
636:                 for claim_a in stmt_a.quantitative_claims:
637:                     for claim_b in stmt_b.quantitative_claims:
638:                         if self._are_comparable_claims(claim_a, claim_b):
639:                             divergence = self._calculate_numerical_divergence(
640:                                 claim_a,
641:                                 claim_b
642:                             )
643:
644:                             if divergence is not None and divergence > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L632_75", 0.2):
645:                                 # Test estadístico de significancia
646:                                 p_value = self._statistical_significance_test(
647:                                     claim_a,
648:                                     claim_b
649:                                 )
650:
651:                                 if p_value < ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_know
ledge_graph", "auto_param_L639_49", 0.05): # Significancia estadística
652:                                     confidence = self.bayesian_calculator.calculate_posterior(
653:                                         evidence_strength=1 - p_value,
654:                                         observations=2,
655:                                         domain_weight=1.5 # Mayor peso para evidencia numérica
656:                                     )
657:
```

```

658:                     evidence = ContradictionEvidence(
659:                         statement_a=stmt_a,
660:                         statement_b=stmt_b,
661:                         contradiction_type=ContradictionType.NUMERICAL_INCONSISTENCY,
662:                         confidence=confidence,
663:                         severity=min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._bu
ild_knowledge_graph", "auto_param_L651_57", 1.0), divergence),
664:                         semantic_similarity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetec
tor._build_knowledge_graph", "auto_param_L652_64", 0.0),
665:                         logical_conflict_score=divergence,
666:                         temporal_consistency=True,
667:                         numerical_divergence=divergence,
668:                         affected_dimensions=[stmt_a.dimension],
669:                         resolutionSuggestions=self._suggest_resolutions(
670:                             ContradictionType.NUMERICAL_INCONSISTENCY
671:                         ),
672:                         statistical_significance=p_value
673:                     )
674:                     contradictions.append(evidence)
675:
676:             return contradictions
677:
678:     def _detect_temporal_conflicts(
679:         self,
680:         statements: list[PolicyStatement]
681:     ) -> list[ContradictionEvidence]:
682:         """Detecta conflictos temporales usando verificaciÃ³n lÃ³gica"""
683:         contradictions = []
684:
685:         # Filtrar declaraciones con marcadores temporales
686:         temporal_statements = [s for s in statements if s.temporal_markers]
687:
688:         if len(temporal_statements) >= 2:
689:             is_consistent, conflicts = self.temporal_verifier.verify_temporal_consistency(
690:                 temporal_statements
691:             )
692:
693:             for conflict in conflicts:
694:                 stmt_a = conflict['event_a']['statement']
695:                 stmt_b = conflict['event_b']['statement']
696:
697:                 confidence = self.bayesian_calculator.calculate_posterior(
698:                     evidence_strength=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph
", "auto_param_L686_38", 0.9), # Alta confianza en lÃ³gica temporal
699:                     observations=len(conflicts),
700:                     domain_weight=1.2
701:                 )
702:
703:                 evidence = ContradictionEvidence(
704:                     statement_a=stmt_a,
705:                     statement_b=stmt_b,
706:                     contradiction_type=ContradictionType.TEMPORAL_CONFLICT,
707:                     confidence=confidence,
708:                     severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_
param_L696_29", 0.8), # Los conflictos temporales son severos
709:                     semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),

```

```

710:         logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_
graph", "auto_param_L698_43", 1.0),
711:             temporal_consistency=False,
712:             numerical_divergence=None,
713:             affected_dimensions=[PolicyDimension.PROGRAMATICO],
714:             resolutionSuggestions=self._suggest_resolutions(
715:                 ContradictionType.TEMPORAL_CONFLICT
716:             )
717:         )
718:     contradictions.append(evidence)
719:
720:     return contradictions
721:
722: def _detect_logical_incompatibilities(
723:     self,
724:     statements: list[PolicyStatement]
725: ) -> list[ContradictionEvidence]:
726:     """Detecta incompatibilidades lógicas usando razonamiento en grafo"""
727:     contradictions = []
728:
729:     # Buscar ciclos negativos en el grafo (indicativos de contradicción)
730:     try:
731:         negative_cycles = nx.negative_edge_cycle(
732:             self.knowledge_graph,
733:             weight='weight'
734:         )
735:
736:         for cycle in negative_cycles:
737:             # Extraer declaraciones del ciclo
738:             stmt_indices = [int(node.split('_')[1]) for node in cycle]
739:             cycle_statements = [statements[i] for i in stmt_indices]
740:
741:             # Analizar incompatibilidad lógica
742:             for i in range(len(cycle_statements)):
743:                 stmt_a = cycle_statements[i]
744:                 stmt_b = cycle_statements[(i + 1) % len(cycle_statements)]
745:
746:                 if self._has_logical_conflict(stmt_a, stmt_b):
747:                     confidence = self.bayesian_calculator.calculate_posterior(
748:                         evidence_strength=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowled
ge_graph", "auto_param_L736_46", 0.85),
749:                         observations=len(cycle),
750:                         domain_weight = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowle
_dge_graph", "domain_weight", 1.0) # Refactored
751:                     )
752:
753:                     evidence = ContradictionEvidence(
754:                         statement_a=stmt_a,
755:                         statement_b=stmt_b,
756:                         contradiction_type=ContradictionType.LOGICAL_INCOMPATIBILITY,
757:                         confidence=confidence,
758:                         severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph"
, "auto_param_L746_37", 0.7),
759:                         semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
760:                         logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_kn
owledge_graph", "auto_param_L748_51", 0.9),

```



```
816:             severity=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledg
e_graph", "auto_param_L804_45", 0.9), # Conflictos de recursos son crÃ-ticos
817:             semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
818:             logical_conflict_score=ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._
build_knowledge_graph", "auto_param_L806_59", 0.8),
819:             temporal_consistency=True,
820:             numerical_divergence=abs(amount_a - amount_b) / max(amount_a, amount_b),
821:             affected_dimensions=[PolicyDimension.FINANCIERO],
822:             resolution_suggestions=self._suggest_resolutions(
823:                 ContradictionType.RESOURCE_ALLOCATION_MISMATCH
824:             )
825:         )
826:         contradictions.append(evidence)
827:
828:     return contradictions
829:
830: def _calculate_coherence_metrics(
831:     self,
832:     contradictions: list[ContradictionEvidence],
833:     statements: list[PolicyStatement],
834:     text: str
835: ) -> dict[str, float]:
836:     """Calcula mÃ©tricas avanzadas de coherencia del documento"""
837:
838:     # Densidad de contradicciones normalizada
839:     contradiction_density = len(contradictions) / max(1, len(statements))
840:
841:     # Ãndice de coherencia semÃ¡ntica global
842:     semantic_coherence = self._calculate_global_semantic_coherence(statements)
843:
844:     # Consistencia temporal
845:     temporal_consistency = sum(
846:         1 for c in contradictions
847:         if c.contradiction_type != ContradictionType.TEMPORAL_CONFLICT
848:     ) / max(1, len(contradictions))
849:
850:     # AlineaciÃ³n de objetivos
851:     objective_alignment = self._calculate_objective_alignment(statements)
852:
853:     # Ãndice de fragmentaciÃ³n del grafo
854:     graph_fragmentation = self._calculate_graph_fragmentation()
855:
856:     # Score de coherencia compuesto (weighted harmonic mean)
857:     weights = np.array([ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_p
aram_L845_28", 0.3), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_33",
0.25), ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_39", 0.2), Paramete
rLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_44", 0.15), ParameterLoaderV2.g
et("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L845_50", 0.1)])
858:     scores = np.array([
859:         1 - contradiction_density,
860:         semantic_coherence,
861:         temporal_consistency,
862:         objective_alignment,
863:         1 - graph_fragmentation
864:     ])
865:
```

```
866:         # Harmonic mean ponderada para penalizar valores bajos
867:         coherence_score = np.sum(weights) / np.sum(weights / np.maximum(scores, ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyCo
ntradiccionDetector._build_knowledge_graph", "auto_param_L855_80", 0.01)))
868:
869:         # Entropía de contradicciones
870:         contradiction_entropy = self._calculate_contradiction_entropy(contradictions)
871:
872:         # Complejidad sintáctica del documento
873:         syntactic_complexity = self._calculate_syntactic_complexity(text)
874:
875:         return {
876:             "coherence_score": float(coherence_score),
877:             "contradiction_density": float(contradiction_density),
878:             "semantic_coherence": float(semantic_coherence),
879:             "temporal_consistency": float(temporal_consistency),
880:             "objective_alignment": float(objective_alignment),
881:             "graph_fragmentation": float(graph_fragmentation),
882:             "contradiction_entropy": float(contradiction_entropy),
883:             "syntactic_complexity": float(syntactic_complexity),
884:             "confidence_interval": self._calculate_confidence_interval(coherence_score, len(statements))
885:         }
886:
887:     def _calculate_global_semantic_coherence(
888:         self,
889:         statements: list[PolicyStatement]
890:     ) -> float:
891:         """Calcula coherencia semántica global usando embeddings"""
892:         if len(statements) < 2:
893:             return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L881
_19", 1.0)
894:
895:         # Calcular matriz de similitud
896:         embeddings = [s.embedding for s in statements if s.embedding is not None]
897:         if len(embeddings) < 2:
898:             return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L886
_19", 0.5)
899:
900:         similarity_matrix = cosine_similarity(embeddings)
901:
902:         # Calcular coherencia como promedio de similitudes consecutivas
903:         consecutive_similarities = []
904:         for i in range(len(similarity_matrix) - 1):
905:             consecutive_similarities.append(similarity_matrix[i, i + 1])
906:
907:         # Penalizar alta varianza en similitudes
908:         mean_similarity = np.mean(consecutive_similarities)
909:         std_similarity = np.std(consecutive_similarities)
910:
911:         coherence = mean_similarity * (1 - min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowle
dge_graph", "auto_param_L899_47", 0.5), std_similarity))
912:
913:         return float(coherence)
914:
915:     def _calculate_objective_alignment(
916:         self,
917:         statements: list[PolicyStatement]
```

```

918:     ) -> float:
919:         """Calcula alineaciÃ³n entre objetivos declarados"""
920:         objective_statements = [
921:             s for s in statements
922:             if s.semantic_role in ['objective', 'goal', 'target']
923:         ]
924:
925:         if len(objective_statements) < 2:
926:             return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L914
_, 1.0)
927:
928:         # Analizar consistencia direccional de objetivos
929:         alignment_scores = []
930:         for i, obj_a in enumerate(objective_statements):
931:             for obj_b in objective_statements[i + 1:]:
932:                 if obj_a.embedding is not None and obj_b.embedding is not None:
933:                     # Calcular alineaciÃ³n como similitud coseno
934:                     alignment = 1 - cosine(obj_a.embedding, obj_b.embedding)
935:                     alignment_scores.append(alignment)
936:
937:         if alignment_scores:
938:             return float(np.mean(alignment_scores))
939:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph", "auto_param_L927_15"
, 0.5)
940:
941:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation")
942:     def _calculate_graph_fragmentation(self) -> float:
943:         """Calcula fragmentaciÃ³n del grafo de conocimiento"""
944:         if self.knowledge_graph.number_of_nodes() == 0:
945:             return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation", "auto_pa
ram_L933_19", 0.0)
946:
947:         # Calcular nÃºmero de componentes conectados
948:         num_components = nx.number_weakly_connected_components(self.knowledge_graph)
949:         num_nodes = self.knowledge_graph.number_of_nodes()
950:
951:         # FragmentaciÃ³n normalizada
952:         fragmentation = (num_components - 1) / max(1, num_nodes - 1)
953:
954:         return float(fragmentation)
955:
956:     def _calculate_contradiction_entropy(
957:         self,
958:         contradictions: list[ContradictionEvidence]
959:     ) -> float:
960:         """Calcula entropÃ-a de distribuciÃ³n de tipos de contradicciÃ³n"""
961:         if not contradictions:
962:             return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_graph_fragmentation", "auto_pa
ram_L950_19", 0.0)
963:
964:         # Contar frecuencia de cada tipo
965:         type_counts = {}
966:         for c in contradictions:
967:             type_counts[c.contradiction_type] = type_counts.get(c.contradiction_type, 0) + 1
968:
969:         # Calcular probabilidades

```

```

970:         total = len(contradictions)
971:         probabilities = [count / total for count in type_counts.values()]
972:
973:         # Calcular entropía de Shannon
974:         entropy = -sum(p * np.log2(p) if p > 0 else 0 for p in probabilities)
975:
976:         # Normalizar por entropía máxima
977:         max_entropy = np.log2(len(ContradictionType))
978:         normalized_entropy = entropy / max_entropy if max_entropy > 0 else 0
979:
980:     return float(normalized_entropy)
981:
982: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity")
983: def _calculate_syntactic_complexity(self, text: str) -> float:
984:     """Calcula complejidad sintáctica del documento"""
985:     doc = self.nlp(text[:5000])  # Limitar para eficiencia
986:
987:     # Matrículas de complejidad
988:     avg_sentence_length = np.mean([len(sent.text.split()) for sent in doc.sents])
989:
990:     # Profundidad promedio del árbol de dependencias
991:     dependency_depths = []
992:     for sent in doc.sents:
993:         depths = [self._get_dependency_depth(token) for token in sent]
994:         if depths:
995:             dependency_depths.append(np.mean(depths))
996:
997:     avg_dependency_depth = np.mean(dependency_depths) if dependency_depths else 0
998:
999:     # Diversidad lingüística (Type-Token Ratio)
1000:    tokens = [token.text.lower() for token in doc if token.is_alpha]
1001:    ttr = len(set(tokens)) / len(tokens) if tokens else 0
1002:
1003:    # Combinar matrículas
1004:    complexity = (
1005:        min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L993_20"), 1.0), avg_sentence_length / 50) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L993_53"), 0.3) +
1006:        min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L994_20"), 1.0), avg_dependency_depth / 10) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L994_54"), 0.3) +
1007:        ttr * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity", "auto_param_L995_22"), 0.4)
1008:
1009:    return float(complexity)
1010:
1011:
1012: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth")
1013: def _get_dependency_depth(self, token) -> int:
1014:     """Calcula profundidad de un token en el árbol de dependencias"""
1015:     depth = 0
1016:     current = token
1017:     while current.head != current and depth < 20:  # Evitar loops infinitos
1018:         current = current.head
1019:         depth += 1
1020:     return depth

```

```
1021:  
1022:     def _calculate_confidence_interval(  
1023:         self,  
1024:         score: float,  
1025:         n_observations: int  
1026:     ) -> tuple[float, float]:  
1027:         """Calcula intervalo de confianza del 95% para el score"""  
1028:         # Usar distribuciÃ³n t de Student para muestras pequeÃ±as  
1029:         if n_observations < 30:  
1030:             # Error estÃ¡ndar estimado  
1031:             se = np.sqrt(score * (1 - score) / n_observations)  
1032:             # Valor crÃ¡tico t para 95% de confianza  
1033:             t_critical = stats.t.ppf(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth",  
"auto_param_L1021_37", 0.975), n_observations - 1)  
1034:             margin = t_critical * se  
1035:         else:  
1036:             # Usar distribuciÃ³n normal para muestras grandes  
1037:             se = np.sqrt(score * (1 - score) / n_observations)  
1038:             margin = 1.96 * se  
1039:  
1040:         return (  
1041:             max(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth", "auto_param_L1029_16  
", 0.0), score - margin),  
1042:             min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_dependency_depth", "auto_param_L1030_16  
", 1.0), score + margin)  
1043:         )  
1044:  
1045:     def _generate_resolution_recommendations(  
1046:         self,  
1047:         contradictions: list[ContradictionEvidence]  
1048:     ) -> list[dict[str, Any]]:  
1049:         """Genera recomendaciones especÃ¡ficas para resolver contradicciones"""  
1050:         recommendations = []  
1051:  
1052:         # Agrupar contradicciones por tipo  
1053:         by_type = {}  
1054:         for c in contradictions:  
1055:             if c.contradiction_type not in by_type:  
1056:                 by_type[c.contradiction_type] = []  
1057:                 by_type[c.contradiction_type].append(c)  
1058:  
1059:         # Generar recomendaciones por tipo  
1060:         for cont_type, conflicts in by_type.items():  
1061:             if cont_type == ContradictionType.NUMERICAL_INCONSISTENCY:  
1062:                 recommendations.append({  
1063:                     "type": "numerical_reconciliation",  
1064:                     "priority": "high",  
1065:                     "description": "Revisar y reconciliar cifras inconsistentes",  
1066:                     "specific_actions": [  
1067:                         "Verificar fuentes de datos originales",  
1068:                         "Establecer lÃ¡nea base Ã³nica",  
1069:                         "Documentar metodologÃ-a de cÃ¡lculo"  
1070:                     ],  
1071:                     "affected_sections": self._identify_affected_sections(conflicts)  
1072:                 })  
1073:
```

```
1074:         elif cont_type == ContradictionType.TEMPORAL_CONFLICT:
1075:             recommendations.append({
1076:                 "type": "timeline_adjustment",
1077:                 "priority": "high",
1078:                 "description": "Ajustar cronograma para resolver conflictos temporales",
1079:                 "specific_actions": [
1080:                     "Revisar secuencia de actividades",
1081:                     "Validar plazos con áreas responsables",
1082:                     "Establecer hitos intermedios claros"
1083:                 ],
1084:                 "affected_sections": self._identify_affected_sections(conflicts)
1085:             })
1086:
1087:             elif cont_type == ContradictionType.RESOURCE_ALLOCATION_MISMATCH:
1088:                 recommendations.append({
1089:                     "type": "budget_reallocation",
1090:                     "priority": "critical",
1091:                     "description": "Revisar asignación presupuestal",
1092:                     "specific_actions": [
1093:                         "Realizar análisis de suficiencia presupuestal",
1094:                         "Priorizar programas según impacto",
1095:                         "Identificar fuentes alternativas de financiación"
1096:                     ],
1097:                     "affected_sections": self._identify_affected_sections(conflicts)
1098:                 })
1099:
1100:             elif cont_type == ContradictionType.SEMANTIC_OPPOSITION:
1101:                 recommendations.append({
1102:                     "type": "conceptual_clarification",
1103:                     "priority": "medium",
1104:                     "description": "Clarificar conceptos y objetivos opuestos",
1105:                     "specific_actions": [
1106:                         "Realizar sesiones de alineación estratégica",
1107:                         "Definir glosario de términos unificado",
1108:                         "Establecer jerarquía clara de objetivos"
1109:                     ],
1110:                     "affected_sections": self._identify_affected_sections(conflicts)
1111:                 })
1112:
1113:             # Ordenar por prioridad
1114:             priority_order = {"critical": 0, "high": 1, "medium": 2, "low": 3}
1115:             recommendations.sort(key=lambda x: priority_order.get(x["priority"], 4))
1116:
1117:             return recommendations
1118:
1119:     def _identify_affected_sections(
1120:         self,
1121:         conflicts: list[ContradictionEvidence]
1122:     ) -> list[str]:
1123:         """Identifica secciones del plan afectadas por contradicciones"""
1124:         affected = set()
1125:         for c in conflicts:
1126:             # Extraer información de sección desde el contexto
1127:             for pattern_name, pattern in self.pdm_patterns.items():
1128:                 if pattern.search(c.statement_a.context_window):
1129:                     affected.add(pattern_name)
```

```

1130:             if pattern.search(c.statement_b.context_window):
1131:                 affected.add(pattern_name)
1132:
1133:         return list(affected)
1134:
1135:     def _serialize_contradiction(
1136:         self,
1137:         contradiction: ContradictionEvidence
1138:     ) -> dict[str, Any]:
1139:         """Serializa evidencia de contradicciÃ³n para output"""
1140:         return {
1141:             "statement_1": contradiction.statement_a.text,
1142:             "statement_2": contradiction.statement_b.text,
1143:             "position_1": contradiction.statement_a.position,
1144:             "position_2": contradiction.statement_b.position,
1145:             "contradiction_type": contradiction.contradiction_type.name,
1146:             "confidence": float(contradiction.confidence),
1147:             "severity": float(contradiction.severity),
1148:             "semantic_similarity": float(contradiction.semantic_similarity),
1149:             "logical_conflict_score": float(contradiction.logical_conflict_score),
1150:             "temporal_consistency": contradiction.temporal_consistency,
1151:             "numerical_divergence": float(
1152:                 contradiction.numerical_divergence) if contradiction.numerical_divergence else None,
1153:             "statistical_significance": float(
1154:                 contradiction.statistical_significance) if contradiction.statistical_significance else None,
1155:             "affected_dimensions": [d.value for d in contradiction.affected_dimensions],
1156:             "resolution_suggestions": contradiction.resolution_suggestions,
1157:             "graph_path": contradiction.graph_path
1158:         }
1159:
1160:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_graph_statistics")
1161:     def _get_graph_statistics(self) -> dict[str, Any]:
1162:         """Obtiene estadÃ¡sticas del grafo de conocimiento"""
1163:         if self.knowledge_graph.number_of_nodes() == 0:
1164:             return {"nodes": 0, "edges": 0, "components": 0}
1165:
1166:         return {
1167:             "nodes": self.knowledge_graph.number_of_nodes(),
1168:             "edges": self.knowledge_graph.number_of_edges(),
1169:             "components": nx.number_weakly_connected_components(self.knowledge_graph),
1170:             "density": nx.density(self.knowledge_graph),
1171:             "average_clustering": nx.average_clustering(self.knowledge_graph.to_undirected()),
1172:             "diameter": nx.diameter(self.knowledge_graph.to_undirected()) if nx.is_connected(
1173:                 self.knowledge_graph.to_undirected()) else -1
1174:         }
1175:
1176:     # MÃ©todos auxiliares
1177:
1178:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_temporal_markers")
1179:     def _extract_temporal_markers(self, text: str) -> list[str]:
1180:         """Extrae marcadores temporales del texto"""
1181:         markers = []
1182:
1183:         # Patrones de fechas
1184:         date_patterns = [
1185:             r'\d{1,2}\s+de\s+\w+\s+de\s+\d{4}',


```

```

1186:             r'\d{4}-\d{2}-\d{2}'',
1187:             r'(enero|febrero|marzo|abril|mayo|junio|julio|agosto|septiembre|octubre|noviembre|diciembre)\s+\d{4}'',
1188:             r'(Q[1-4]|trimestre\s+[1-4])\s+\d{4}'',
1189:             r'20\d{2}'',
1190:             r'(corto|mediano|largo)\s+plazo',
1191:             r'(primer|segundo|tercer|cuarto)\s+(aÃ±o|semestre|trimestre)'
1192:         ]
1193:
1194:     for pattern in date_patterns:
1195:         matches = re.findall(pattern, text, re.IGNORECASE)
1196:         markers.extend(matches)
1197:
1198:     return markers
1199:
1200: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_quantitative_claims")
1201: def _extract_quantitative_claims(self, text: str) -> list[dict[str, Any]]:
1202:     """Extrae afirmaciones cuantitativas estructuradas"""
1203:     claims = []
1204:
1205:     # Patrones numÃ©ricos con contexto
1206:     patterns = [
1207:         r'(\d+(?:[.,]\d+)?)\s*(%|por\s*ciento)', 'percentage'),
1208:         r'(\d+(?:[.,]\d+)?)\s*(millones?|mil\s+millones?)', 'amount'),
1209:         r'(\$|COP)\s*(\d+(?:[.,]\d+)?', 'currency'),
1210:         r'(\d+(?:[.,]\d+)?)\s*(personas?|beneficiarios?|familias?)', 'beneficiaries'),
1211:         r'(\d+(?:[.,]\d+)?)\s*(hectÃ;reas?|km2?|metros?)', 'area'),
1212:         r'meta\s+de\s+(\d+(?:[.,]\d+)?', 'target')
1213:     ]
1214:
1215:     for pattern, claim_type in patterns:
1216:         matches = re.finditer(pattern, text, re.IGNORECASE)
1217:         for match in matches:
1218:             value_str = match.group(1) if claim_type != 'currency' else match.group(2)
1219:             value = self._parse_number(value_str)
1220:
1221:             claims.append({
1222:                 'type': claim_type,
1223:                 'value': value,
1224:                 'raw_text': match.group(0),
1225:                 'position': match.span(),
1226:                 'context': text[max(0, match.start() - 20):min(len(text), match.end() + 20)]
1227:             })
1228:
1229:     return claims
1230:
1231: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._parse_number")
1232: def _parse_number(self, text: str) -> float:
1233:     """Parsea nÃºmero desde texto"""
1234:     try:
1235:         # Reemplazar coma decimal
1236:         normalized = text.replace(',', '.')
1237:         return float(normalized)
1238:     except ValueError:
1239:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._parse_number", "auto_param_L1227_19", 0.
0)
1240:

```

```

1241:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._extract_resource_mentions")
1242:     def _extract_resource_mentions(self, text: str) -> list[tuple[str, float | None]]:
1243:         """Extrae menciones de recursos con montos"""
1244:         resources = []
1245:
1246:         # Patrones de recursos específicos de PDM colombiano
1247:         resource_patterns = [
1248:             r'SGP\s*[:\s]*\$?\s*(\d+(:[.,]\d+))\s*(millones?)?', 'SGP'),
1249:             r'regalÃ-as\s*[:\s]*\$?\s*(\d+(:[.,]\d+))\s*(millones?)?', 'regalÃ-as'),
1250:             r'reursos\s+proprios\s*[:\s]*\$?\s*(\d+(:[.,]\d+))\s*(millones?)?', 'recursos_propios'),
1251:             r'cofinaciÃ³n\s*[:\s]*\$?\s*(\d+(:[.,]\d+))\s*(millones?)?', 'cofinaciaciÃ³n'),
1252:             r'presupuesto\s+total\s*[:\s]*\$?\s*(\d+(:[.,]\d+))\s*(millones?)?', 'presupuesto_total')
1253:         ]
1254:
1255:         for pattern, resource_type in resource_patterns:
1256:             matches = re.finditer(pattern, text, re.IGNORECASE)
1257:             for match in matches:
1258:                 amount = self._parse_number(match.group(1)) if match.group(1) else None
1259:                 if match.group(2) and 'millon' in match.group(2).lower():
1260:                     amount = amount * 1000000 if amount else None
1261:                 resources.append((resource_type, amount))
1262:
1263:         return resources
1264:
1265:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._determine_semantic_role")
1266:     def _determine_semantic_role(self, sent) -> str | None:
1267:         """Determina el rol semántico de una oración"""
1268:         # Safely extract text (handles both strings and spacy objects)
1269:         text_lower = safe_text_extract(sent).lower()
1270:
1271:         role_patterns = {
1272:             'objective': ['objetivo', 'meta', 'propÃ³sito', 'finalidad'],
1273:             'strategy': ['estrategia', 'lÃnea', 'eje', 'pilar'],
1274:             'action': ['implementar', 'ejecutar', 'desarrollar', 'realizar'],
1275:             'indicator': ['indicador', 'medir', 'evaluar', 'monitorear'],
1276:             'resource': ['presupuesto', 'recurso', 'financiaciÃ³n', 'inversiÃ³n'],
1277:             'constraint': ['limitaciÃ³n', 'restricciÃ³n', 'condiciÃ³n', 'requisito']
1278:         }
1279:
1280:         for role, keywords in role_patterns.items():
1281:             if any(keyword in text_lower for keyword in keywords):
1282:                 return role
1283:
1284:         return None
1285:
1286:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._identify_dependencies")
1287:     def _identify_dependencies(self, sent, doc) -> set[str]:
1288:         """Identifica dependencias entre declaraciones"""
1289:         dependencies = set()
1290:
1291:         # Buscar referencias a otras secciones
1292:         reference_patterns = [
1293:             r'como\s+se\s+mciona\s+en',
1294:             r'segÃ³n\s+lo\s+establecido\s+en',
1295:             r'de\s+acuerdo\s+con',
1296:             r'en\s+lÃ-nea\s+con',

```

```
1297:         r'siguiendo\s+lo\s+dispuesto'
1298:     ]
1299:
1300:     for pattern in reference_patterns:
1301:         if re.search(pattern, sent.text, re.IGNORECASE):
1302:             # Buscar la secciÃ³n referenciada
1303:             for other_sent in doc.sents:
1304:                 if other_sent != sent:
1305:                     # Usar hash de los primeros 50 caracteres como ID
1306:                     dependencies.add(other_sent.text[:50])
1307:
1308:     return dependencies
1309:
1310: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_context_window")
1311: def _get_context_window(self, text: str, start: int, end: int, window_size: int = 200) -> str:
1312:     """Obtiene ventana de contexto alrededor de una posiciÃ³n"""
1313:     context_start = max(0, start - window_size)
1314:     context_end = min(len(text), end + window_size)
1315:     return text[context_start:context_end]
1316:
1317: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_similarity")
1318: def _calculate_similarity(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) -> float:
1319:     """Calcula similaridad entre dos declaraciones"""
1320:     if stmt_a.embedding is not None and stmt_b.embedding is not None:
1321:         return float(1 - cosine(stmt_a.embedding, stmt_b.embedding))
1322:     return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._calculate_similarity", "auto_param_L1310_15"
, 0.0)
1323:
1324: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction")
1325: def _classify_contradiction(self, text: str) -> float:
1326:     """Clasifica probabilidad de contradicciÃ³n en texto"""
1327:     try:
1328:         result = self.contradiction_classifier(text)
1329:         # Buscar score de contradicciÃ³n
1330:         for item in result:
1331:             if 'contradiction' in item['label'].lower():
1332:                 return item['score']
1333:     return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction", "auto_param_L13
21_19", 0.0)
1334:     except Exception as e:
1335:         logger.warning(f"Error en clasificaciÃ³n de contradicciÃ³n: {e}")
1336:     return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._classify_contradiction", "auto_param_L13
24_19", 0.0)
1337:
1338: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight")
1339: def _get_domain_weight(self, dimension: PolicyDimension) -> float:
1340:     """Obtiene peso especÃ¢fico del dominio"""
1341:     weights = {
1342:         PolicyDimension.DIAGNOSTICO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight"
, "auto_param_L1330_41", 0.8),
1343:         PolicyDimension.ESTRATEGICO: 1.2,
1344:         PolicyDimension.PROGRAMATICO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight"
, "auto_param_L1332_42", 1.0),
1345:         PolicyDimension.FINANCIERO: 1.5,
1346:         PolicyDimension.SEGUIMIENTO: ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight"
, "auto_param_L1334_41", 0.9),
```

```

1347:         PolicyDimension.TERRITORIAL: 1.1
1348:     }
1349:     return weights.get(dimension, ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._get_domain_weight", "auto_param_L1337_38", 1.0))
1350:
1351:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._suggest_resolutions")
1352:     def _suggest_resolutions(self, contradiction_type: ContradictionType) -> list[str]:
1353:         """Sugiere resoluciones especÃ¡ficas por tipo de contradicciÃ³n"""
1354:         suggestions = {
1355:             ContradictionType.NUMERICAL_INCONSISTENCY: [
1356:                 "Verificar fuentes de datos y metodologÃ-as de cÃ¡lculo",
1357:                 "Establecer lÃnea base Ã³nica con validaciÃ³n tÃ©cnica",
1358:                 "Documentar supuestos y proyecciones utilizadas"
1359:             ],
1360:             ContradictionType.TEMPORAL_CONFLICT: [
1361:                 "Revisar cronograma maestro del plan",
1362:                 "Validar secuencia lÃ³gica de actividades",
1363:                 "Ajustar plazos segÃºn capacidad institucional"
1364:             ],
1365:             ContradictionType.SEMANTIC_OPPOSITION: [
1366:                 "Realizar taller de alineaciÃ³n conceptual",
1367:                 "Clarificar definiciones en glosario tÃ©cnico",
1368:                 "Priorizar objetivos segÃºn Plan Nacional de Desarrollo"
1369:             ],
1370:             ContradictionType.RESOURCE_ALLOCATION_MISMATCH: [
1371:                 "Realizar anÃ¡lisis de brechas financieras",
1372:                 "Priorizar inversiones segÃºn impacto social",
1373:                 "Explorar fuentes alternativas de financiaciÃ³n"
1374:             ],
1375:             ContradictionType.LOGICAL_INCOMPATIBILITY: [
1376:                 "Revisar cadena de valor de programas",
1377:                 "Validar teorÃ-a de cambio del plan",
1378:                 "Eliminar duplicidades y solapamientos"
1379:             ]
1380:         }
1381:         return suggestions.get(contradiction_type, ["Revisar y ajustar segÃºn contexto"])
1382:
1383:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._are_comparable_claims")
1384:     def _are_comparable_claims(self, claim_a: dict, claim_b: dict) -> bool:
1385:         """Determina si dos afirmaciones cuantitativas son comparables"""
1386:         # Mismo tipo y contexto similar
1387:         if claim_a['type'] != claim_b['type']:
1388:             return False
1389:
1390:         # Verificar si hablan del mismo concepto
1391:         context_similarity = self._text_similarity(
1392:             claim_a.get('context', ''),
1393:             claim_b.get('context', '')
1394:         )
1395:
1396:         return context_similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._are_comparable_claims", "auto_param_L1384_36", 0.6)
1397:
1398:     @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity")
1399:     def _text_similarity(self, text_a: str, text_b: str) -> float:
1400:         """Calcula similaridad simple entre textos"""

```

```
1401:     if not text_a or not text_b:
1402:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1390_19",
1403: 0.0)
1404: 
1405:     # TokenizaciÃ³n simple
1406:     tokens_a = set(text_a.lower().split())
1407:     tokens_b = set(text_b.lower().split())
1408: 
1409:     if not tokens_a or not tokens_b:
1410:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1397_19",
1411: 0.0)
1412: 
1413:     # Coeficiente de Jaccard
1414:     intersection = tokens_a & tokens_b
1415:     union = tokens_a | tokens_b
1416: 
1417:     return len(intersection) / len(union) if union else ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1403_60", 0.0)
1418: 
1419: def _calculate_numerical_divergence(
1420:     self,
1421:     claim_a: dict,
1422:     claim_b: dict
1423: ) -> float | None:
1424:     """Calcula divergencia entre valores numÃ©ricos"""
1425:     value_a = claim_a.get('value', 0)
1426:     value_b = claim_b.get('value', 0)
1427: 
1428:     if value_a == 0 and value_b == 0:
1429:         return None
1430: 
1431:     # Divergencia relativa
1432:     max_value = max(abs(value_a), abs(value_b))
1433:     if max_value == 0:
1434:         return None
1435: 
1436:     divergence = abs(value_a - value_b) / max_value
1437:     return divergence
1438: 
1439: def _statistical_significance_test(
1440:     self,
1441:     claim_a: dict,
1442:     claim_b: dict
1443: ) -> float:
1444:     """Realiza test de significancia estadÃ-stica"""
1445:     value_a = claim_a.get('value', 0)
1446:     value_b = claim_b.get('value', 0)
1447: 
1448:     # Test t de una muestra para diferencia significativa
1449:     # Asumiendo distribuciÃ³n normal con varianza estimada
1450:     diff = abs(value_a - value_b)
1451:     pooled_value = (value_a + value_b) / 2
1452: 
1453:     if pooled_value == 0:
1454:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1440_19",
1.0) # No significativo
```

```
1453:
1454:     # EstimaciÃ³n conservadora de error estÃ¡ndar
1455:     se = pooled_value * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L
1443_28", 0.1) # 10% de error estimado
1456:
1457:     if se == 0:
1458:         return ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity", "auto_param_L1446_19",
0.0) # Altamente significativo
1459:
1460:     # EstadÃ-stico t
1461:     t_stat = diff / se
1462:
1463:     # Valor p aproximado (two-tailed)
1464:     p_value = 2 * (1 - stats.norm.cdf(abs(t_stat)))
1465:
1466:     return p_value
1467:
1468: @calibrated_method("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict")
1469: def _has_logical_conflict(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) -> bool:
1470:     """Determina si hay conflicto lÃ³gico entre declaraciones"""
1471:     # Verificar si las declaraciones tienen roles incompatibles
1472:     if stmt_a.semantic_role and stmt_b.semantic_role:
1473:         incompatible_roles = [
1474:             ('objective', 'constraint'),
1475:             ('strategy', 'constraint'),
1476:             ('action', 'constraint')
1477:         ]
1478:
1479:         for role_pair in incompatible_roles:
1480:             if (stmt_a.semantic_role in role_pair and
1481:                 stmt_b.semantic_role in role_pair and
1482:                 stmt_a.semantic_role != stmt_b.semantic_role):
1483:                 return True
1484:
1485:     # Verificar negaciÃ³n explÃ- cita
1486:     negation_patterns = ['no', 'nunca', 'ningÃ³n', 'sin', 'tampoco']
1487:     has_negation_a = any(pattern in stmt_a.text.lower() for pattern in negation_patterns)
1488:     has_negation_b = any(pattern in stmt_b.text.lower() for pattern in negation_patterns)
1489:
1490:     # Si una tiene negaciÃ³n y otra no, y son similares, hay conflicto
1491:     if has_negation_a != has_negation_b:
1492:         similarity = self._calculate_similarity(stmt_a, stmt_b)
1493:         if similarity > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_pa
ram_L1481_28", 0.7):
1494:             return True
1495:
1496:     return False
1497:
1498: def _are_conflicting_allocations(
1499:     self,
1500:     amount_a: float,
1501:     amount_b: float,
1502:     total: float
1503: ) -> bool:
1504:     """Determina si las asignaciones de recursos estÃ¡n en conflicto"""
1505:     # Si la suma excede el total disponible
```

```

1506:         if amount_a + amount_b > total * 1.1: # 10% de margen
1507:             return True
1508:
1509:         # Si hay una diferencia muy grande entre asignaciones similares
1510:         return abs(amount_a - amount_b) / max(amount_a, amount_b) > ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1498_68", 0.5)
1511:
1512:     def _determine_relation_type(
1513:         self,
1514:         stmt_a: PolicyStatement,
1515:         stmt_b: PolicyStatement
1516:     ) -> str:
1517:         """Determina el tipo de relaciÃ³n entre dos declaraciones"""
1518:         # Analizar roles semÃ¡nticos
1519:         if stmt_a.semantic_role and stmt_b.semantic_role:
1520:             if stmt_a.semantic_role == stmt_b.semantic_role:
1521:                 return "parallel"
1522:             elif stmt_a.semantic_role in ["strategy", "objective"] and stmt_b.semantic_role == "action":
1523:                 return "enables"
1524:             elif stmt_a.semantic_role == "action" and stmt_b.semantic_role in ["indicator", "resource"]:
1525:                 return "requires"
1526:
1527:         # Analizar dependencias
1528:         if stmt_a.dependencies & {stmt_b.text[:50]}:
1529:             return "depends_on"
1530:
1531:         # Por defecto, relaciÃ³n de similaridad
1532:         return "related"
1533:
1534:     def _calculate_severity(
1535:         self,
1536:         stmt_a: PolicyStatement,
1537:         stmt_b: PolicyStatement
1538:     ) -> float:
1539:         """Calcula la severidad de una contradicciÃ³n entre declaraciones"""
1540:         severity = ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "severity", 0.5)
# Refactored
1541:
1542:         # Incrementar si las declaraciones estÃ¡n en la misma dimensiÃ³n
1543:         if stmt_a.dimension == stmt_b.dimension:
1544:             severity += ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1532_24", 0.2)
1545:
1546:         # Incrementar si tienen muchas entidades en comÃºn
1547:         common_entities = set(stmt_a.entities) & set(stmt_b.entities)
1548:         if len(common_entities) > 0:
1549:             severity += min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1537_28", 0.2), len(common_entities) * ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1537_56", 0.05))
1550:
1551:         # Incrementar si tienen marcadores temporales en conflicto
1552:         if stmt_a.temporal_markers and stmt_b.temporal_markers:
1553:             severity += ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1541_24", 0.1)
1554:
1555:         return min(ParameterLoaderV2.get("farfan_core.analysis.contradiction_deteccion.PolicyContradictionDetector._has_logical_conflict", "auto_param_L1543

```

```
_19", 1.0), severity)
1556:
1557:
1558:
1559: =====
1560: FILE: src/farfan_pipeline/analysis/derek_beach.py
1561: =====
1562:
1563: #!/usr/bin/env python3
1564: """
1565: Causal Deconstruction and Audit Framework (CDAF) v2.0
1566: Framework de ProducciÃ³n para AnÃ¡lisis Causal de Planes de Desarrollo Territorial
1567:
1568: THEORETICAL FOUNDATION (Derek Beach):
1569: "A causal mechanism is a system of interlocking parts (entities engaging in
1570: activities) that transmits causal forces from X to Y" (Beach 2016: 465)
1571:
1572: This framework implements Theory-Testing Process Tracing with mechanistic evidence
1573: evaluation using Beach's evidential tests taxonomy (Beach & Pedersen 2019).
1574:
1575: Author: AI Systems Architect
1576: Version: 2.0.0 (Beach-Grounded Production Grade)
1577: """
1578:
1579: import argparse
1580: import hashlib
1581: import json
1582: import logging
1583: import re
1584: import sys
1585: import warnings
1586: from collections import defaultdict
1587: from dataclasses import asdict, dataclass, field
1588: from pathlib import Path
1589: from typing import (
1590:     TYPE_CHECKING,
1591:     Any,
1592:     Literal,
1593:     NamedTuple,
1594:     TypedDict,
1595:     cast,
1596: )
1597: from farfan_pipeline.core.parameters import ParameterLoaderV2
1598: from farfan_pipeline.core.calibration.decorators import calibrated_method
1599:
1600: if TYPE_CHECKING:
1601:     import fitz
1602:
1603: # Core dependencies
1604: try:
1605:     import networkx as nx
1606:     import numpy as np
1607:     import pandas as pd
1608:     import spacy
1609:     import yaml
1610:     from fuzzywuzzy import fuzz, process
```

```
1611:     from pydantic import BaseModel, Field, ValidationError, validator
1612:     from pydot import Dot, Edge, Node
1613:     from scipy.spatial.distance import cosine
1614:     from scipy.special import rel_entr
1615: except ImportError as e:
1616:     print(f"ERROR: Dependencia faltante. Ejecute: pip install {e.name}")
1617:     sys.exit(1)
1618:
1619: # DNP Standards Integration
1620: try:
1621:     from dnp_integration import ValidadorDNP
1622:
1623:     DNP_AVAILABLE = True
1624: except ImportError:
1625:     DNP_AVAILABLE = False
1626:     warnings.warn("Mádulos DNP no disponibles. Validación DNP desabilitada.", stacklevel=2)
1627:
1628: # Refactored Bayesian Engine (F1.2: Architectural Refactoring)
1629: try:
1630:     from inference.bayesian_adapter import BayesianEngineAdapter
1631:
1632:     REFACTORED_BAYESIAN_AVAILABLE = True
1633: except ImportError:
1634:     REFACTORED_BAYESIAN_AVAILABLE = False
1635:     warnings.warn("Motor Bayesiano refactorizado no disponible. Usando implementación legacy.", stacklevel=2)
1636:
1637: # Configure logging
1638: logging.basicConfig(
1639:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
1640:     datefmt='%Y-%m-%d %H:%M:%S'
1641: )
1642:
1643: # =====
1644: # CONSTANTS
1645: # =====
1646: DEFAULT_CONFIG_FILE = "config.yaml"
1647: EXTRACTION_REPORT_SUFFIX = "_extraction_confidence_report.json"
1648: CAUSAL_MODEL_SUFFIX = "_causal_model.json"
1649: DNP_REPORT_SUFFIX = "_dnp_compliance_report.txt"
1650:
1651: # Type definitions
1652: NodeType = Literal["programa", "producto", "resultado", "impacto"]
1653: RigorStatus = Literal["fuerte", "dÁobil", "sin_evaluar"]
1654: TestType = Literal["hoop_test", "smoking_gun", "doubly_decisive", "straw_in_wind"]
1655: DynamicsType = Literal["suma", "decreciente", "constante", "indefinido"]
1656:
1657: # =====
1658: # BEACH THEORETICAL PRIMITIVES - Added to existing code
1659: # =====
1660:
1661: class BeachEvidentialTest:
1662:     """
1663:         Derek Beach evidential tests implementation (Beach & Pedersen 2019: Ch 5).
1664:
1665:         FOUR-FOLD TYPOLOGY calibrated by necessity (N) and sufficiency (S):
1666:
```

```
1667: HOOP TEST [N: High, S: Low]:
1668: - Fail → 206\222 ELIMINATES hypothesis (definitive knock-out)
1669: - Pass → 206\222 Hypothesis survives but not proven
1670: - Example: "Responsible entity must be documented"
1671:
1672: SMOKING GUN [N: Low, S: High]:
1673: - Pass → 206\222 Strongly confirms hypothesis
1674: - Fail → 206\222 Doesn't eliminate (could be false negative)
1675: - Example: "Unique policy instrument only used for this mechanism"
1676:
1677: DOUBLY DECISIVE [N: High, S: High]:
1678: - Pass → 206\222 Conclusively confirms
1679: - Fail → 206\222 Conclusively eliminates
1680: - Extremely rare in social science
1681:
1682: STRAW-IN-WIND [N: Low, S: Low]:
1683: - Pass/Fail → 206\222 Marginal confidence change
1684: - Used for preliminary screening
1685:
1686: REFERENCE: Beach & Pedersen (2019), pp 117-126
1687: """
1688:
1689: @staticmethod
1690: def classify_test(necessity: float, sufficiency: float) -> TestType:
1691:     """
1692:         Classify evidential test type based on necessity and sufficiency.
1693:
1694:         Beach calibration:
1695:             - Necessity > 0.7 → 206\222 High necessity
1696:             - Sufficiency > 0.7 → 206\222 High sufficiency
1697:     """
1698:     high_n = necessity > 0.7
1699:     high_s = sufficiency > 0.7
1700:
1701:     if high_n and high_s:
1702:         return "doubly_decisional"
1703:     elif high_n and not high_s:
1704:         return "hoop_test"
1705:     elif not high_n and high_s:
1706:         return "smoking_gun"
1707:     else:
1708:         return "straw_in_wind"
1709:
1710: @staticmethod
1711: def apply_test_logic(test_type: TestType, evidence_found: bool,
1712:                      prior: float, bayes_factor: float) -> tuple[float, str]:
1713:     """
1714:         Apply Beach test-specific logic to Bayesian updating.
1715:
1716:         CRITICAL RULES:
1717:             1. Hoop Test FAIL → 206\222 posterior → 211\210 0 (knock-out)
1718:             2. Smoking Gun PASS → 206\222 multiply prior by large BF (>10)
1719:             3. Doubly Decisive → 206\222 extreme updates (BF > 100 or < 0.01)
1720:
1721:         Returns: (posterior_confidence, interpretation)
1722:     """
```

```

1723:     if test_type == "hoop_test":
1724:         if not evidence_found:
1725:             # KNOCK-OUT per Beach: "hypothesis must jump through hoop"
1726:             return 0.01, "HOOP_TEST_FAILURE: Hypothesis eliminated"
1727:         else:
1728:             # Pass: necessary condition met, use standard Bayesian
1729:             posterior = min(0.95, prior * bayes_factor)
1730:             return posterior, "HOOP_TEST_PASSED: Hypothesis survives, not proven"
1731:
1732:     elif test_type == "smoking_gun":
1733:         if evidence_found:
1734:             # Strong confirmation: unique evidence found
1735:             posterior = min(0.98, prior * max(bayes_factor, 10.0))
1736:             return posterior, "SMOKING_GUN_FOUND: Strong confirmation"
1737:         else:
1738:             # Doesn't eliminate: could be false negative
1739:             posterior = prior * 0.9 # slight penalty
1740:             return posterior, "SMOKING_GUN_NOT_FOUND: Doesn't eliminate"
1741:
1742:     elif test_type == "doubly_decisional":
1743:         if evidence_found:
1744:             return 0.99, "DOUBLY_DECISIONAL_CONFIRMED: Conclusive"
1745:         else:
1746:             return 0.01, "DOUBLY_DECISIONAL_ELIMINATED: Conclusive"
1747:
1748:     # Marginal update only
1749:     elif evidence_found:
1750:         posterior = min(0.95, prior * min(bayes_factor, 2.0))
1751:         return posterior, "STRAW_IN_WIND: Weak support"
1752:     else:
1753:         posterior = max(0.05, prior / min(bayes_factor, 2.0))
1754:         return posterior, "STRAW_IN_WIND: Weak disconfirmation"
1755:
1756: # =====
1757: # Custom Exceptions - Structured Error Semantics
1758: # =====
1759:
1760: class CDAFException(Exception):
1761:     """Base exception for CDAF framework with structured payloads"""
1762:
1763:     def __init__(self, message: str, details: dict[str, Any] | None = None,
1764:                  stage: str | None = None, recoverable: bool = False) -> None:
1765:         self.message = message
1766:         self.details = details or {}
1767:         self.stage = stage
1768:         self.recoverable = recoverable
1769:         super().__init__(self._format_message())
1770:
1771:     @calibrated_method("farfan_core.analysis.derek_beach.CDAFException._format_message")
1772:     def _format_message(self) -> str:
1773:         """Format error message with structured information"""
1774:         parts = ["[CDAF Error]"]
1775:         if self.stage:
1776:             parts.append(f"[Stage: {self.stage}]")
1777:         parts.append(self.message)
1778:         if self.details:

```

```
1779:         parts.append(f"Details: {json.dumps(self.details, indent=2)}")
1780:     return " ".join(parts)
1781:
1782:     @calibrated_method("farfan_core.analysis.derek_beach.CDAFException.to_dict")
1783:     def to_dict(self) -> dict[str, Any]:
1784:         """Convert exception to structured dictionary"""
1785:         return {
1786:             'error_type': self.__class__.__name__,
1787:             'message': self.message,
1788:             'details': self.details,
1789:             'stage': self.stage,
1790:             'recoverable': self.recoverable
1791:         }
1792:
1793: class CDAFValidationError(CDAFException):
1794:     """Configuration or data validation error"""
1795:     pass
1796:
1797: class CDAFProcessingError(CDAFException):
1798:     """Error during document processing"""
1799:     pass
1800:
1801: class CDAFBayesianError(CDAFException):
1802:     """Error during Bayesian inference"""
1803:     pass
1804:
1805: class CDAFConfigError(CDAFException):
1806:     """Configuration loading or validation error"""
1807:     pass
1808:
1809: # =====
1810: # Pydantic Configuration Models - Schema Validation at Load Time
1811: # =====
1812:
1813: class BayesianThresholdsConfig(BaseModel):
1814:     """Bayesian inference thresholds configuration"""
1815:     kl_divergence: float = Field(
1816:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "kl_divergence", 0.01),
1817:         ge=0.0,
1818:         le=1.0,
1819:         description="KL divergence threshold for convergence"
1820:     )
1821:     convergence_min_evidence: int = Field(
1822:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "convergence_min_evidence", 2),
1823:         ge=1,
1824:         description="Minimum evidence count for convergence check"
1825:     )
1826:     prior_alpha: float = Field(
1827:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "prior_alpha", 2.0),
1828:         ge=0.1,
1829:         description="Default alpha parameter for Beta prior"
1830:     )
1831:     prior_beta: float = Field(
1832:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "prior_beta", 2.0),
1833:         ge=0.1,
1834:         description="Default beta parameter for Beta prior"
```

```
1835:     )
1836:     laplace_smoothing: float = Field(
1837:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianThresholdsConfig", "laplace_smoothing", 1.0),
1838:         ge=0.0,
1839:         description="Laplace smoothing parameter"
1840:     )
1841:
1842: class MechanismTypeConfig(BaseModel):
1843:     """Mechanism type prior probabilities"""
1844:     administrativo: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "administrativo", 0.30), ge=0.0, le=
1.0)
1845:     tecnico: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "tecnico", 0.25), ge=0.0, le=1.0)
1846:     financiero: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "financiero", 0.20), ge=0.0, le=1.0)
1847:     politico: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "politico", 0.15), ge=0.0, le=1.0)
1848:     mixto: float = Field(default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismTypeConfig", "mixto", 0.10), ge=0.0, le=1.0)
1849:
1850:     @validator('*',
1851:               pre=True, always=True)
1852:     def check_sum_to_one(cls, v, values):
1853:         """Validate that probabilities sum to approximately 1.0"""
1854:         if len(values) == 4: # All fields loaded
1855:             total = sum(values.values()) + v
1856:             if abs(total - 1.0) > 0.01:
1857:                 raise ValueError(f"Mechanism type priors must sum to 1.0, got {total}")
1858:
1859: class PerformanceConfig(BaseModel):
1860:     """Performance and optimization settings"""
1861:     enable_vectorized_ops: bool = Field(
1862:         default=True,
1863:         description="Use vectorized numpy operations where possible"
1864:     )
1865:     enable_async_processing: bool = Field(
1866:         default=False,
1867:         description="Enable async processing for large PDFs (experimental)"
1868:     )
1869:     max_context_length: int = Field(
1870:         default=1000,
1871:         ge=100,
1872:         description="Maximum context length for spaCy processing"
1873:     )
1874:     cache_embeddings: bool = Field(
1875:         default=True,
1876:         description="Cache spaCy embeddings for reuse"
1877:     )
1878:
1879: class SelfReflectionConfig(BaseModel):
1880:     """Self-reflective learning configuration"""
1881:     enable_prior_learning: bool = Field(
1882:         default=False,
1883:         description="Enable learning from audit feedback to update priors"
1884:     )
1885:     feedback_weight: float = Field(
1886:         default=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.SelfReflectionConfig", "feedback_weight", 0.1),
1887:         ge=0.0,
1888:         le=1.0,
1889:         description="Weight for feedback in prior updates (0=ignore, 1=full)"
```

```
1890:     )
1891:     prior_history_path: str | None = Field(
1892:         default=None,
1893:         description="Path to save/load historical priors"
1894:     )
1895:     min_documents_for_learning: int = Field(
1896:         default=5,
1897:         ge=1,
1898:         description="Minimum documents before applying learned priors"
1899:     )
1900:
1901: class CDAFConfigSchema(BaseModel):
1902:     """Complete CDAF configuration schema with validation"""
1903:     patterns: dict[str, str] = Field(
1904:         description="Regex patterns for document parsing"
1905:     )
1906:     lexicons: dict[str, Any] = Field(
1907:         description="Lexicons for causal logic, classification, etc."
1908:     )
1909:     entity_aliases: dict[str, str] = Field(
1910:         description="Entity name aliases and mappings"
1911:     )
1912:     verb_sequences: dict[str, int] = Field(
1913:         description="Verb sequence ordering for temporal coherence"
1914:     )
1915:     bayesian_thresholds: BayesianThresholdsConfig = Field(
1916:         default_factory=BayesianThresholdsConfig,
1917:         description="Bayesian inference thresholds"
1918:     )
1919:     mechanism_type_priors: MechanismTypeConfig = Field(
1920:         default_factory=MechanismTypeConfig,
1921:         description="Prior probabilities for mechanism types"
1922:     )
1923:     performance: PerformanceConfig = Field(
1924:         default_factory=PerformanceConfig,
1925:         description="Performance and optimization settings"
1926:     )
1927:     self_reflection: SelfReflectionConfig = Field(
1928:         default_factory=SelfReflectionConfig,
1929:         description="Self-reflective learning configuration"
1930:     )
1931:
1932:     class Config:
1933:         extra = 'allow' # Allow additional fields for extensibility
1934:
1935: class GoalClassification(NamedTuple):
1936:     """Classification structure for goals"""
1937:     type: NodeType
1938:     dynamics: DynamicsType
1939:     test_type: TestType
1940:     confidence: float
1941:
1942: class EntityActivity(NamedTuple):
1943:     """
1944:     Entity-Activity tuple for mechanism parts (Beach 2016).
1945:
```

```
1946:     BEACH DEFINITION:
1947:     "A mechanism part consists of an entity (organization, actor, structure)
1948:     engaging in an activity that transmits causal forces" (Beach 2016: 465)
1949:
1950:     This is the FUNDAMENTAL UNIT of mechanistic evidence in Process Tracing.
1951:     """
1952:     entity: str
1953:     activity: str
1954:     verb_lemma: str
1955:     confidence: float
1956:
1957: class CausalLink(TypedDict):
1958:     """Structure for causal links in the graph"""
1959:     source: str
1960:     target: str
1961:     logic: str
1962:     strength: float
1963:     evidence: list[str]
1964:     posterior_mean: float | None
1965:     posterior_std: float | None
1966:     kl_divergence: float | None
1967:     converged: bool | None
1968:
1969: class AuditResult(TypedDict):
1970:     """Audit result structure"""
1971:     passed: bool
1972:     warnings: list[str]
1973:     errors: list[str]
1974:     recommendations: list[str]
1975:
1976: @dataclass
1977: class MetaNode:
1978:     """Comprehensive node structure for goals/metas"""
1979:     id: str
1980:     text: str
1981:     type: NodeType
1982:     baseline: float | str | None = None
1983:     target: float | str | None = None
1984:     unit: str | None = None
1985:     responsible_entity: str | None = None
1986:     entity_activity: EntityActivity | None = None
1987:     financial_allocation: float | None = None
1988:     unit_cost: float | None = None
1989:     rigor_status: RigorStatus = "sin_evaluar"
1990:     dynamics: DynamicsType = "indefinido"
1991:     test_type: TestType = "straw_in_wind"
1992:     contextual_risks: list[str] = field(default_factory=list)
1993:     causal_justification: list[str] = field(default_factory=list)
1994:     audit_flags: list[str] = field(default_factory=list)
1995:     confidence_score: float = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MetaNode", "confidence_score", 0.0)
1996:
1997: class ConfigLoader:
1998:     """External configuration management with Pydantic schema validation"""
1999:
2000:     def __init__(self, config_path: Path) -> None:
2001:         self.logger = logging.getLogger(self.__class__.__name__)
```

```

2002:     self.config_path = config_path
2003:     self.config: dict[str, Any] = {}
2004:     self.validated_config: CDAFConfigSchema | None = None
2005:     # HARMONIC FRONT 4: Track uncertainty over iterations
2006:     self._uncertainty_history: list[float] = []
2007:     self._load_config()
2008:     self._validate_config()
2009:     self._load_uncertainty_history()
2010:
2011:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_config")
2012:     def _load_config(self) -> None:
2013:         """Load YAML configuration file"""
2014:         try:
2015:             with open(self.config_path, 'r', encoding='utf-8') as f:
2016:                 self.config = yaml.safe_load(f)
2017:                 self.logger.info(f"ConfiguraciÃ³n cargada desde {self.config_path}")
2018:         except FileNotFoundError:
2019:             self.logger.warning(f"Archivo de configuraciÃ³n no encontrado: {self.config_path}")
2020:             self._load_default_config()
2021:         except Exception as e:
2022:             raise CDAFConfigError(
2023:                 "Error cargando configuraciÃ³n",
2024:                 details={'path': str(self.config_path), 'error': str(e)},
2025:                 stage="config_load",
2026:                 recoverable=True
2027:             )
2028:
2029:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config")
2030:     def _load_default_config(self) -> None:
2031:         """Load default configuration if custom fails"""
2032:         self.config = {
2033:             'patterns': {
2034:                 'section_titles': r'^(:CAPÃ\215TULO|ARTÃ\215CULO|PARTE)\s+[\dIVX]+',
2035:                 'goal_codes': r'[MP][RIP]-\d{3}',
2036:                 'numeric_formats': r'[\d.]+\d+.%?',
2037:                 'table_headers': r'(:PROGRAMA|META|INDICADOR|LÃ\215NEA BASE|VALOR ESPERADO)',
2038:                 'financial_headers': r'(:PRESUPUESTO|VALOR|MONTO|INVERSIÃ\223N)'
2039:             },
2040:             'lexicons': {
2041:                 'causal_logic': [
2042:                     'gracias a', 'con el fin de', 'para lograr', 'mediante',
2043:                     'a travÃ©s de', 'como resultado de', 'debido a', 'porque',
2044:                     'por medio de', 'permitirÃ; ', 'contribuirÃ; a'
2045:                 ],
2046:                 'goal_classification': {
2047:                     'tasa': 'decreciente',
2048:                     'Ã-ndice': 'constante',
2049:                     'nÃºmero': 'suma',
2050:                     'porcentaje': 'constante',
2051:                     'cantidad': 'suma',
2052:                     'cobertura': 'suma'
2053:                 },
2054:                 'contextual_factors': [
2055:                     'riesgo', 'amenaza', 'obstÃ;culo', 'limitaciÃ³n',
2056:                     'restricciÃ³n', 'desafÃ;o', 'brecha', 'dÃ©ficit',
2057:                     'vulnerabilidad', 'hipÃ³tesis alternativa'

```

```

2058:         ],
2059:         'administrative_keywords': [
2060:             'gestiÃ³n', 'administraciÃ³n', 'coordinaciÃ³n', 'regulaciÃ³n',
2061:             'normativa', 'institucional', 'gobernanza', 'reglamento',
2062:             'decreto', 'resoluciÃ³n', 'acuerdo'
2063:         ]
2064:     },
2065:     'entity_aliases': {
2066:         'SEC GOB': 'SecretarÃ-a de Gobierno',
2067:         'SEC PLAN': 'SecretarÃ-a de PlaneaciÃ³n',
2068:         'SEC HAC': 'SecretarÃ-a de Hacienda',
2069:         'SEC SALUD': 'SecretarÃ-a de Salud',
2070:         'SEC EDU': 'SecretarÃ-a de EducaciÃ³n',
2071:         'SEC INFRA': 'SecretarÃ-a de Infraestructura'
2072:     },
2073:     'verb_sequences': {
2074:         'diagnosticar': 1,
2075:         'identificar': 2,
2076:         'analizar': 3,
2077:         'diseÃ±ar': 4,
2078:         'planificar': 5,
2079:         'implementar': 6,
2080:         'ejecutar': 7,
2081:         'monitorear': 8,
2082:         'evaluar': 9
2083:     },
2084:     # Bayesian thresholds - now externalized
2085:     'bayesian_thresholds': {
2086:         'kl_divergence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "kl_divergence", 0.01),
2087:         'convergence_min_evidence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "convergence_min_evi
dence", 2),
2088:         'prior_alpha': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "prior_alpha", 2.0),
2089:         'prior_beta': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "prior_beta", 2.0),
2090:         'laplace_smoothing': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "laplace_smoothing", 1.0)
2091:     },
2092:     # Mechanism type priors - now externalized
2093:     'mechanism_type_priors': {
2094:         'administrativo': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "administrativo", 0.30),
2095:         'tecnico': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "tecnico", 0.25),
2096:         'financiero': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "financiero", 0.20),
2097:         'politico': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "politico", 0.15),
2098:         'mixto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "mixto", 0.10)
2099:     },
2100:     # Performance settings
2101:     'performance': {
2102:         'enable_vectorized_ops': True,
2103:         'enable_async_processing': False,
2104:         'max_context_length': 1000,
2105:         'cache_embeddings': True
2106:     },
2107:     # Self-reflection settings
2108:     'self_reflection': {
2109:         'enable_prior_learning': False,
2110:         'feedback_weight': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader._load_default_config", "feedback_weight", 0.1),
2111:         'prior_history_path': None,
2112:         'min_documents_for_learning': 5

```

```
2113:         }
2114:     }
2115:     self.logger.warning("Usando configuraciÃ³n por defecto")
2116:
2117:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._validate_config")
2118:     def _validate_config(self) -> None:
2119:         """Validate configuration structure using Pydantic schema"""
2120:         try:
2121:             # Validate with Pydantic schema
2122:             self.validated_config = CDAFConfigSchema(**self.config)
2123:             self.logger.info("â\234\223 ConfiguraciÃ³n validada exitosamente con esquema Pydantic")
2124:         except ValidationError as e:
2125:             error_details = {
2126:                 'validation_errors': [
2127:                     {
2128:                         'field': '.'.join(str(x) for x in err['loc']),
2129:                         'error': err['msg'],
2130:                         'type': err['type']
2131:                     }
2132:                     for err in e.errors()
2133:                 ]
2134:             }
2135:             raise CDAFValidationError(
2136:                 "ConfiguraciÃ³n invÃ¡lida - errores de esquema",
2137:                 details=error_details,
2138:                 stage="config_validation",
2139:                 recoverable=False
2140:             )
2141:
2142:     # Legacy validation for required sections
2143:     required_sections = ['patterns', 'lexicons', 'entity_aliases', 'verb_sequences']
2144:     for section in required_sections:
2145:         if section not in self.config:
2146:             self.logger.warning(f"SecciÃ³n faltante en configuraciÃ³n: {section}")
2147:             self.config[section] = {}
2148:
2149:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get")
2150:     def get(self, key: str, default: Any = None) -> Any:
2151:         """Get configuration value with dot notation support"""
2152:         keys = key.split('.')
2153:         value = self.config
2154:         for k in keys:
2155:             if isinstance(value, dict):
2156:                 value = value.get(k, default)
2157:             else:
2158:                 return default
2159:         return value
2160:
2161:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_bayesian_threshold")
2162:     def get_bayesian_threshold(self, key: str) -> float:
2163:         """Get Bayesian threshold with type safety"""
2164:         if self.validated_config:
2165:             return getattr(self.validated_config.bayesian_thresholds, key)
2166:         return self.get(f'bayesian_thresholds.{key}', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_bayesian_threshold", "default
", 0.01))
2167:
```

```

2168:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior")
2169:     def get_mechanism_prior(self, mechanism_type: str) -> float:
2170:         """Get mechanism type prior probability with type safety"""
2171:         if self.validated_config:
2172:             return getattr(self.validated_config.mechanism_type_priors, mechanism_type, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior", "default", 0.0))
2173:         return self.get(f'mechanism_type_priors.{mechanism_type}', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.get_mechanism_prior", "default", 0.0))
2174:
2175:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.get_performance_setting")
2176:     def get_performance_setting(self, key: str) -> Any:
2177:         """Get performance setting with type safety"""
2178:         if self.validated_config:
2179:             return getattr(self.validated_config.performance, key)
2180:         return self.get(f'performance.{key}')
2181:
2182:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback")
2183:     def update_priors_from_feedback(self, feedback_data: dict[str, Any]) -> None:
2184:         """
2185:             Self-reflective loop: Update priors based on audit feedback
2186:             Implements frontier paradigm of learning from results
2187:
2188:             HARMONIC FRONT 4 ENHANCEMENT:
2189:             - Applies penalties to mechanism types with implementation_failure flags
2190:             - Heavily penalizes "miracle" mechanisms failing necessity/sufficiency tests
2191:             - Ensures mean mech_uncertainty decreases by ~11% over iterations
2192:         """
2193:         if not self.validated_config or not self.validated_config.self_reflection.enable_prior_learning:
2194:             self.logger.debug("Prior learning disabled")
2195:             return
2196:
2197:         feedback_weight = self.validated_config.self_reflection.feedback_weight
2198:
2199:         # Track initial priors for uncertainty measurement
2200:         initial_priors = {}
2201:         for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
2202:             if hasattr(self.validated_config.mechanism_type_priors, attr):
2203:                 initial_priors[attr] = getattr(self.validated_config.mechanism_type_priors, attr)
2204:
2205:         # Update mechanism type priors based on observed frequencies
2206:         if 'mechanism_frequencies' in feedback_data:
2207:             for mech_type, observed_freq in feedback_data['mechanism_frequencies'].items():
2208:                 if hasattr(self.validated_config.mechanism_type_priors, mech_type):
2209:                     current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
2210:                     # Weighted update: new_prior = (1-weight)*current + weight*observed
2211:                     updated_prior = (1 - feedback_weight) * current_prior + feedback_weight * observed_freq
2212:                     setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
2213:                     self.config['mechanism_type_priors'][mech_type] = updated_prior
2214:
2215:         # NEW: Apply penalty factors for failing mechanism types
2216:         if 'penalty_factors' in feedback_data:
2217:             penalty_weight = feedback_weight * 1.5 # Heavier penalty than positive feedback
2218:             for mech_type, penalty_factor in feedback_data['penalty_factors'].items():
2219:                 if hasattr(self.validated_config.mechanism_type_priors, mech_type):
2220:                     current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
2221:                     # Apply penalty: reduce prior for frequently failing types

```

```

2222:             penalized_prior = current_prior * penalty_factor
2223:             # Blend with current
2224:             updated_prior = (1 - penalty_weight) * current_prior + penalty_weight * penalized_prior
2225:             setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
2226:             self.config['mechanism_type_priors'][mech_type] = updated_prior
2227:             self.logger.info(f"Applied penalty to {mech_type}: {current_prior:.4f} -> {updated_prior:.4f}")
2228:
2229:             # NEW: Heavy penalty for "miracle" mechanisms failing necessity/sufficiency
2230:             test_failures = feedback_data.get('test_failures', {})
2231:             if test_failures.get('necessity_failures', 0) > 0 or test_failures.get('sufficiency_failures', 0) > 0:
2232:                 # If failures exist, apply additional penalty to 'politico' (often "miracle" type)
2233:                 # and 'mixto' (vague mechanism types)
2234:                 miracle_types = ['politico', 'mixto']
2235:                 miracle_penalty = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback", "miracle_penalty", 0.85) #
Refactored
2236:             for mech_type in miracle_types:
2237:                 if hasattr(self.validated_config.mechanism_type_priors, mech_type):
2238:                     current_prior = getattr(self.validated_config.mechanism_type_priors, mech_type)
2239:                     updated_prior = current_prior * miracle_penalty
2240:                     setattr(self.validated_config.mechanism_type_priors, mech_type, updated_prior)
2241:                     self.config['mechanism_type_priors'][mech_type] = updated_prior
2242:                     self.logger.info(
2243:                         f"Miracle mechanism penalty for {mech_type}: {current_prior:.4f} -> {updated_prior:.4f}")
2244:
2245:             # Renormalize to ensure priors sum to ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.update_priors_from_feedback", "auto_param
L686_46", 1.0)
2246:             total_prior = sum(
2247:                 getattr(self.validated_config.mechanism_type_priors, attr)
2248:                 for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']
2249:                 if hasattr(self.validated_config.mechanism_type_priors, attr)
2250:             )
2251:
2252:             if total_prior > 0:
2253:                 for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
2254:                     if hasattr(self.validated_config.mechanism_type_priors, attr):
2255:                         current = getattr(self.validated_config.mechanism_type_priors, attr)
2256:                         normalized = current / total_prior
2257:                         setattr(self.validated_config.mechanism_type_priors, attr, normalized)
2258:                         self.config['mechanism_type_priors'][attr] = normalized
2259:
2260:             # Calculate uncertainty reduction for quality criteria
2261:             final_priors = {}
2262:             for attr in ['administrativo', 'tecnico', 'financiero', 'politico', 'mixto']:
2263:                 if hasattr(self.validated_config.mechanism_type_priors, attr):
2264:                     final_priors[attr] = getattr(self.validated_config.mechanism_type_priors, attr)
2265:
2266:             # Calculate entropy as uncertainty measure
2267:             initial_entropy = -sum(p * np.log(p + 1e-10) for p in initial_priors.values() if p > 0)
2268:             final_entropy = -sum(p * np.log(p + 1e-10) for p in final_priors.values() if p > 0)
2269:             uncertainty_reduction = ((initial_entropy - final_entropy) / max(initial_entropy, 1e-10)) * 100
2270:
2271:             self.logger.info(f"Uncertainty reduction: {uncertainty_reduction:.2f}%")
2272:
2273:             # Save updated priors if history path configured
2274:             if self.validated_config.self_reflection.prior_history_path:
2275:                 self._save_prior_history(feedback_data, uncertainty_reduction)

```

```
2276:         self.logger.info(f"Prioras actualizados con peso de retroalimentaciÃ³n {feedback_weight}")
2277: 
2278:     @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._save_prior_history")
2279:     def _save_prior_history(self, feedback_data: dict[str, Any] | None = None,
2280:                            uncertainty_reduction: float | None = None) -> None:
2281:         """
2282:         Save prior history for learning across documents
2283: 
2284:         HARMONIC FRONT 4 ENHANCEMENT:
2285:         - Tracks uncertainty reduction over iterations
2286:         - Records penalty applications and test failures
2287:         """
2288: 
2289:     if not self.validated_config or not self.validated_config.self_reflection.prior_history_path:
2290:         return
2291: 
2292:     try:
2293:         history_path = Path(self.validated_config.self_reflection.prior_history_path)
2294:         history_path.parent.mkdir(parents=True, exist_ok=True)
2295: 
2296:         # Load existing history if available
2297:         history_records = []
2298:         if history_path.exists():
2299:             try:
2300:                 with open(history_path, 'r', encoding='utf-8') as f:
2301:                     existing_data = json.load(f)
2302:                     if isinstance(existing_data, list):
2303:                         history_records = existing_data
2304:                     elif isinstance(existing_data, dict) and 'history' in existing_data:
2305:                         history_records = existing_data['history']
2306:             except json.JSONDecodeError:
2307:                 self.logger.warning("Existing history file corrupted, starting fresh")
2308: 
2309:             # Create new record
2310:             history_record = {
2311:                 'mechanism_type_priors': dict(self.config.get('mechanism_type_priors', {})),
2312:                 'timestamp': pd.Timestamp.now().isoformat(),
2313:                 'version': '2.0'
2314:             }
2315: 
2316:             # Add feedback metrics if available
2317:             if feedback_data:
2318:                 history_record['audit_quality'] = feedback_data.get('audit_quality', {})
2319:                 history_record['test_failures'] = feedback_data.get('test_failures', {})
2320:                 history_record['penalty_factors'] = feedback_data.get('penalty_factors', {})
2321: 
2322:             if uncertainty_reduction is not None:
2323:                 history_record['uncertainty_reduction_percent'] = uncertainty_reduction
2324: 
2325:             history_records.append(history_record)
2326: 
2327:             # Save complete history
2328:             history_data = {
2329:                 'version': '2.0',
2330:                 'harmonic_front': 4,
2331:                 'last_updated': pd.Timestamp.now().isoformat(),
```

```
2332:             'total_iterations': len(history_records),
2333:             'history': history_records
2334:         }
2335:
2336:         with open(history_path, 'w', encoding='utf-8') as f:
2337:             json.dump(history_data, f, indent=2)
2338:
2339:             self.logger.info(f"Historial de priors guardado en {history_path} (iteraciÃ³n {len(history_records)} )")
2340:         except Exception as e:
2341:             self.logger.warning(f"Error guardando historial de priors: {e}")
2342:
2343: @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader._load_uncertainty_history")
2344: def _load_uncertainty_history(self) -> None:
2345:     """
2346:     Load historical uncertainty measurements
2347:
2348:     HARMONIC FRONT 4: Required for tracking \211% reduction over 10 iterations
2349:     """
2350:     if not self.validated_config or not self.validated_config.self_reflection.prior_history_path:
2351:         return
2352:
2353:     try:
2354:         history_path = Path(self.validated_config.self_reflection.prior_history_path)
2355:         if history_path.exists():
2356:             with open(history_path, 'r', encoding='utf-8') as f:
2357:                 history_data = json.load(f)
2358:                 if isinstance(history_data, dict) and 'history' in history_data:
2359:                     # Extract uncertainty from each record
2360:                     for record in history_data['history']:
2361:                         if 'uncertainty_reduction_percent' in record:
2362:                             self._uncertainty_history.append(
2363:                                 record['uncertainty_reduction_percent']
2364:                             )
2365:             self.logger.info(f"Loaded {len(self._uncertainty_history)} uncertainty measurements")
2366:         except Exception as e:
2367:             self.logger.warning(f"Could not load uncertainty history: {e}")
2368:
2369: @calibrated_method("farfan_core.analysis.derek_beach.ConfigLoader.check_uncertainty_reduction_criterion")
2370: def check_uncertainty_reduction_criterion(self, current_uncertainty: float) -> dict[str, Any]:
2371:     """
2372:     Check if mean mechanism_type uncertainty has decreased \211% over 10 iterations
2373:
2374:     HARMONIC FRONT 4 QUALITY CRITERIA:
2375:     Success verified if mean mech_uncertainty decreases by \211% over 10 sequential PDM analyses
2376:     """
2377:     self._uncertainty_history.append(current_uncertainty)
2378:
2379:     # Keep only last 10 iterations
2380:     recent_history = self._uncertainty_history[-10:]
2381:
2382:     result = {
2383:         'current_uncertainty': current_uncertainty,
2384:         'iterations_tracked': len(recent_history),
2385:         'criterion_met': False,
2386:         'reduction_percent': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ConfigLoader.check_uncertainty_reduction_criterion", "reduction_per
cent", 0.0),
```

```
2387:         'status': 'insufficient_data'
2388:     }
2389:
2390:     if len(recent_history) >= 10:
2391:         initial_uncertainty = recent_history[0]
2392:         final_uncertainty = recent_history[-1]
2393:
2394:         if initial_uncertainty > 0:
2395:             reduction_percent = ((initial_uncertainty - final_uncertainty) / initial_uncertainty) * 100
2396:             result['reduction_percent'] = reduction_percent
2397:             result['criterion_met'] = reduction_percent >= 5.0
2398:             result['status'] = 'success' if result['criterion_met'] else 'needs_improvement'
2399:
2400:             self.logger.info(
2401:                 f"Uncertainty reduction over 10 iterations: {reduction_percent:.2f}%" +
2402:                 f"(criterion: {len(recent_history)}%, met: {result['criterion_met']}))"
2403:             )
2404:         else:
2405:             self.logger.info(
2406:                 f"Uncertainty tracking: {len(recent_history)}/10 iterations " +
2407:                 f"(need {10 - len(recent_history)} more for criterion check)"
2408:             )
2409:
2410:     return result
2411:
2412: class PDFProcessor:
2413:     """Advanced PDF processing and extraction"""
2414:
2415:     def __init__(self, config: ConfigLoader, retry_handler=None) -> None:
2416:         self.logger = logging.getLogger(self.__class__.__name__)
2417:         self.config = config
2418:         self.document: fitz.Document | None = None
2419:         self.text_content: str = ""
2420:         self.tables: list[pd.DataFrame] = []
2421:         self.metadata: dict[str, Any] = {}
2422:         self.retry_handler = retry_handler
2423:
2424:     @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.load_document")
2425:     def load_document(self, pdf_path: Path) -> bool:
2426:         """Load PDF document with retry logic"""
2427:         if self.retry_handler:
2428:             try:
2429:                 from farfan_pipeline.analysis.retry_handler import DependencyType
2430:
2431:                 @self.retry_handler.with_retry(
2432:                     DependencyType.PDF_PARSER,
2433:                     operation_name="open_pdf",
2434:                     exceptions=(IOError, OSError, RuntimeError)
2435:                 )
2436:             def load_with_retry():
2437:                 import fitz
2438:                 doc = fitz.open(pdf_path)
2439:                 self.logger.info(f"PDF cargado: {pdf_path.name} ({len(doc)} páginas)")
2440:                 return doc
2441:
2442:             self.document = load_with_retry()
```



```
2499:             header_text = ' '.join(str(cell) for cell in df.iloc[0] if cell)
2500:             if table_pattern.search(header_text):
2501:                 self.tables.append(df)
2502:                 self.logger.info(f"Tabla extraÃ±a de pÃ¡gina {page_num}: {df.shape}")
2503:             except Exception as e:
2504:                 self.logger.warning(f"Error procesando tabla en pÃ¡gina {page_num}: {e}")
2505:             except Exception as e:
2506:                 self.logger.debug(f"Error extrayendo tablas de pÃ¡gina {page_num}: {e}")
2507:
2508:             self.logger.info(f"Total de tablas extraÃ±as: {len(self.tables)}")
2509:             return self.tables
2510:
2511: @calibrated_method("farfan_core.analysis.derek_beach.PDFProcessor.extract_sections")
2512: def extract_sections(self) -> dict[str, str]:
2513:     """Extract document sections based on patterns"""
2514:     sections = {}
2515:     section_pattern = re.compile(
2516:         self.config.get('patterns.section_titles', r'^(?:CAPÃ\215TULO|ARTÃ\215CULO)\s+[\dIVX]+'),
2517:         re.MULTILINE | re.IGNORECASE
2518:     )
2519:
2520:     matches = list(section_pattern.finditer(self.text_content))
2521:
2522:     for i, match in enumerate(matches):
2523:         section_title = match.group().strip()
2524:         start_pos = match.end()
2525:         end_pos = matches[i + 1].start() if i + 1 < len(matches) else len(self.text_content)
2526:         sections[section_title] = self.text_content[start_pos:end_pos].strip()
2527:
2528:     self.logger.info(f"Secciones identificadas: {len(sections)}")
2529:     return sections
2530:
2531: class CausalExtractor:
2532:     """Extract and structure causal chains from text"""
2533:
2534:     def __init__(self, config: ConfigLoader, nlp_model: spacy.Language) -> None:
2535:         self.logger = logging.getLogger(self.__class__.__name__)
2536:         self.config = config
2537:         self.nlp = nlp_model
2538:         self.graph = nx.DiGraph()
2539:         self.nodes: dict[str, MetaNode] = {}
2540:         self.causal_chains: list[CausalLink] = []
2541:
2542: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor.extract_causal_hierarchy")
2543: def extract_causal_hierarchy(self, text: str) -> nx.DiGraph:
2544:     """Extract complete causal hierarchy from text"""
2545:     # Extract goals/metas
2546:     goals = self._extract_goals(text)
2547:
2548:     # Build hierarchy
2549:     for goal in goals:
2550:         self._add_node_to_graph(goal)
2551:
2552:     # Extract causal connections
2553:     self._extract_causal_links(text)
2554:
```

```

2555:     # Build hierarchy based on goal types
2556:     self._build_type_hierarchy()
2557:
2558:     self.logger.info(f"Grafo causal construido: {self.graph.number_of_nodes()} nodos, "
2559:                      f"{self.graph.number_of_edges()} aristas")
2560:     return self.graph
2561:
2562: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_goals")
2563: def _extract_goals(self, text: str) -> list[MetaNode]:
2564:     """Extract all goals from text"""
2565:     goals = []
2566:     goal_pattern = re.compile(
2567:         self.config.get('patterns.goal_codes', r'[MP][RIP]-\d{3}'),
2568:         re.IGNORECASE
2569:     )
2570:
2571:     for match in goal_pattern.finditer(text):
2572:         goal_id = match.group().upper()
2573:         context_start = max(0, match.start() - 500)
2574:         context_end = min(len(text), match.end() + 500)
2575:         context = text[context_start:context_end]
2576:
2577:         goal = self._parse_goal_context(goal_id, context)
2578:         if goal:
2579:             goals.append(goal)
2580:             self.nodes[goal.id] = goal
2581:
2582:     self.logger.info(f"Metas extraídas: {len(goals)}")
2583:     return goals
2584:
2585: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._parse_goal_context")
2586: def _parse_goal_context(self, goal_id: str, context: str) -> MetaNode | None:
2587:     """Parse goal context to extract structured information"""
2588:     # Determine goal type
2589:     if goal_id.startswith('MP'):
2590:         node_type = 'producto'
2591:     elif goal_id.startswith('MR'):
2592:         node_type = 'resultado'
2593:     elif goal_id.startswith('MI'):
2594:         node_type = 'impacto'
2595:     else:
2596:         node_type = 'programa'
2597:
2598:     # Extract numerical values
2599:     numeric_pattern = re.compile(
2600:         self.config.get('patterns.numeric_formats', r'[\d,]+(?:\.\d+)?%?')
2601:     )
2602:     numbers = numeric_pattern.findall(context)
2603:
2604:     # Process with spaCy
2605:     doc = self.nlp(context[:1000])
2606:
2607:     # Extract entities
2608:     entities = [ent.text for ent in doc.ents if ent.label_ in ['ORG', 'PER', 'LOC']]
2609:
2610:     # Create goal node

```

```
2611:     goal = MetaNode(
2612:         id=goal_id,
2613:         text=context[:200].strip(),
2614:         type=cast("NodeType", node_type),
2615:         baseline=numbers[0] if len(numbers) > 0 else None,
2616:         target=numbers[1] if len(numbers) > 1 else None,
2617:         responsible_entity=entities[0] if entities else None
2618:     )
2619:
2620:     return goal
2621:
2622: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_goal_text")
2623: def _extract_goal_text(self, text: str, **kwargs) -> str | None:
2624: """
2625: Extract the text content associated with a specific goal ID.
2626:
2627: This method extracts goal text from the provided document text. It can work
2628: in two modes:
2629: 1. If a goal_id is provided in kwargs, it extracts text for that specific goal
2630: 2. Otherwise, it returns the first goal text found in the document
2631:
2632: Args:
2633:     text: The full document text
2634:     **kwargs: Additional parameters including optional 'goal_id', 'data',
2635:               'sentences', 'tables'
2636:
2637: Returns:
2638:     The extracted text for the goal, or None if not found
2639: """
2640: # Get goal_id from kwargs if provided, otherwise look for data parameter
2641: goal_id = kwargs.get('goal_id')
2642: kwargs.get('data')
2643:
2644: # If no goal_id specified, try to extract the first goal from text
2645: if not goal_id:
2646:     goal_pattern = re.compile(
2647:         r'\b[MP][RIP]-\d{3}\b',
2648:         re.IGNORECASE
2649:     )
2650:     match = goal_pattern.search(text)
2651:     if match:
2652:         goal_id = match.group().upper()
2653:     else:
2654:         # No goal found in text
2655:         return None
2656:
2657: # Now extract the context around the goal_id
2658: goal_pattern = re.compile(
2659:     rf'\b{re.escape(goal_id)}\b',
2660:     re.IGNORECASE
2661: )
2662:
2663: match = goal_pattern.search(text)
2664: if not match:
2665:     return None
2666:
```

```
2667:     # Extract context around the goal ID
2668:     context_start = max(0, match.start() - 500)
2669:     context_end = min(len(text), match.end() + 500)
2670:     context = text[context_start:context_end]
2671:
2672:     return context.strip()
2673:
2674: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._add_node_to_graph")
2675: def _add_node_to_graph(self, node: MetaNode) -> None:
2676:     """Add node to causal graph"""
2677:     node_dict = asdict(node)
2678:     # Convert NamedTuple to dict for JSON serialization
2679:     if node.entity_activity:
2680:         node_dict['entity_activity'] = node.entity_activity._asdict()
2681:     self.graph.add_node(node.id, **node_dict)
2682:
2683: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links")
2684: def _extract_causal_links(self, text: str) -> None:
2685:     """
2686:     AGUJA I: El Prior Informado Adaptativo
2687:     Extract causal links using Bayesian inference with adaptive priors
2688:     """
2689:     causal_keywords = self.config.get('lexicons.causal_logic', [])
2690:
2691:     # Get externalized thresholds from configuration
2692:     kl_threshold = self.config.get_bayesian_threshold('kl_divergence')
2693:     convergence_min_evidence = self.config.get_bayesian_threshold('convergence_min_evidence')
2694:
2695:     # Track evidence for each potential link
2696:     link_evidence: dict[tuple[str, str], list[dict[str, Any]]] = defaultdict(list)
2697:
2698:     # Phase 1: Collect all evidence
2699:     for keyword in causal_keywords:
2700:         pattern = re.compile(
2701:             rf'({"\|".join(re.escape(nid) for nid in self.nodes)})'
2702:             rf'\s+{re.escape(keyword)}\s+'
2703:             rf'({"\|".join(re.escape(nid) for nid in self.nodes)})',
2704:             re.IGNORECASE
2705:         )
2706:
2707:         for match in pattern.finditer(text):
2708:             source = match.group(1).upper()
2709:             target = match.group(2).upper()
2710:             logic = match.group(0)
2711:
2712:             if source in self.nodes and target in self.nodes:
2713:                 # Extract context around the match for language specificity analysis
2714:                 context_start = max(0, match.start() - 100)
2715:                 context_end = min(len(text), match.end() + 100)
2716:                 match_context = text[context_start:context_end]
2717:
2718:                 # Calculate evidence components
2719:                 evidence = {
2720:                     'keyword': keyword,
2721:                     'logic': logic,
2722:                     'match_position': match.start(),
```

```

2723:             'semantic_distance': self._calculate_semantic_distance(source, target),
2724:             'type_transition_prior': self._calculate_type_transition_prior(source, target),
2725:             'language_specificity': self._calculate_language_specificity(keyword, None, match_context),
2726:             'temporal_coherence': self._assess_temporal_coherence(source, target),
2727:             'financial_consistency': self._assess_financial_consistency(source, target),
2728:             'textual_proximity': self._calculate_textual_proximity(source, target, text)
2729:         }
2730:
2731:         link_evidence[(source, target)].append(evidence)
2732:
2733:     # Phase 2: Bayesian inference for each link
2734:     for (source, target), evidences in link_evidence.items():
2735:         # Initialize prior distribution
2736:         prior_mean, prior_alpha, prior_beta = self._initialize_prior(source, target)
2737:
2738:         # Incremental Bayesian update
2739:         posterior_alpha = prior_alpha
2740:         posterior_beta = prior_beta
2741:         kl_divs = []
2742:
2743:         for evidence in evidences:
2744:             # Calculate likelihood components
2745:             likelihood = self._calculate_composite_likelihood(evidence)
2746:
2747:             # Update Beta distribution parameters
2748:             # Using Beta-Binomial conjugate prior
2749:             posterior_alpha += likelihood
2750:             posterior_beta += (1 - likelihood)
2751:
2752:             # Calculate KL divergence for convergence check
2753:             if len(kl_divs) > 0:
2754:                 prior_dist = np.array([posterior_alpha - likelihood, posterior_beta - (1 - likelihood)])
2755:                 prior_dist = prior_dist / prior_dist.sum()
2756:                 posterior_dist = np.array([posterior_alpha, posterior_beta])
2757:                 posterior_dist = posterior_dist / posterior_dist.sum()
2758:                 kl_div = float(np.sum(rel_ent(posterior_dist, prior_dist)))
2759:                 kl_divs.append(kl_div)
2760:
2761:             # Calculate posterior statistics
2762:             posterior_mean = posterior_alpha / (posterior_alpha + posterior_beta)
2763:             posterior_var = (posterior_alpha * posterior_beta) / (
2764:                 (posterior_alpha + posterior_beta) ** 2 * (posterior_alpha + posterior_beta + 1))
2765:             )
2766:             posterior_std = np.sqrt(posterior_var)
2767:
2768:             # AUDIT POINT 2.1: Structural Veto (D6-Q2)
2769:             # TeoriaCambio validation - caps Bayesian posterior at 0.6 for impermissible links
2770:             # Implements axiomatic-Bayesian fusion per Goertz & Mahoney 2012
2771:             structuralViolation = self._check_structuralViolation(source, target)
2772:             if structuralViolation:
2773:                 # Deterministic veto: cap posterior at ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "structural_veto_threshold", 0.6) despite high semantic evidence
2774:                 original_posterior = posterior_mean
2775:                 posterior_mean = min(posterior_mean, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links", "structural_veto_threshold", 0.6))

```

```

2776:             self.logger.warning(
2777:                 f"STRUCTURAL VETO (D6-Q2): Link {source}â\206\222{target} violates causal hierarchy. "
2778:                 f"Posterior capped from {original_posterior:.3f} to {posterior_mean:.3f}. "
2779:                 f"Violation: {structuralViolation}"
2780:             )
2781:
2782:             # Check convergence (require minimum evidence count)
2783:             converged = (len(kl_divs) >= convergence_min_evidence and
2784:                         len(kl_divs) > 0 and kl_divs[-1] < kl_threshold)
2785:             final_kl = kl_divs[-1] if len(kl_divs) > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_links",
2786: "final_kl_default", 0.0)
2787:
2788:             # Add edge with posterior distribution
2789:             self.graph.add_edge(
2790:                 source, target,
2791:                 logic=evidences[0]['logic'],
2792:                 keyword=evidences[0]['keyword'],
2793:                 strength=float(posterior_mean),
2794:                 posterior_mean=float(posterior_mean),
2795:                 posterior_std=float(posterior_std),
2796:                 posterior_alpha=float(posterior_alpha),
2797:                 posterior_beta=float(posterior_beta),
2798:                 kl_divergence=float(final_kl),
2799:                 converged=converged,
2800:                 evidence_count=len(evidences),
2801:                 structuralViolation=structuralViolation,
2802:                 veto_applied=structuralViolation is not None
2803:             )
2804:
2805:             self.causal_chains.append({
2806:                 'source': source,
2807:                 'target': target,
2808:                 'logic': evidences[0]['logic'],
2809:                 'strength': float(posterior_mean),
2810:                 'evidence': [e['keyword'] for e in evidences],
2811:                 'posterior_mean': float(posterior_mean),
2812:                 'posterior_std': float(posterior_std),
2813:                 'kl_divergence': float(final_kl),
2814:                 'converged': converged
2815:             })
2816:
2817:             self.logger.info(f"Enlaces causales extraÃ±os: {len(self.causal_chains)} "
2818:                             f"(con inferencia Bayesiana)")
2819: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance")
2820: def _calculate_semantic_distance(self, source: str, target: str) -> float:
2821:     """
2822:         Calculate semantic distance between nodes using spaCy embeddings
2823:
2824:         PERFORMANCE NOTE: This method can be optimized with:
2825:         1. Vectorized operations using numpy for batch processing
2826:         2. Embedding caching to avoid recomputing spaCy vectors
2827:         3. Async processing for large documents with many nodes
2828:         4. Alternative: BERT/transformer embeddings for higher fidelity (SOTA)
2829:
2830:         Current implementation prioritizes determinism over speed.

```

```

2831:     Enable performance.cache_embeddings in config for production use.
2832:     """
2833:     try:
2834:         source_node = self.nodes.get(source)
2835:         target_node = self.nodes.get(target)
2836:
2837:         if not source_node or not target_node:
2838:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)
2839:
2840:         # TODO: Implement embedding cache if performance.cache_embeddings is enabled
2841:         # This would save ~60% computation time on large documents
2842:
2843:         # Use spaCy to get embeddings
2844:         max_context = self.config.get_performance_setting('max_context_length') or 1000
2845:         source_doc = self.nlp(source_node.text[:max_context])
2846:         target_doc = self.nlp(target_node.text[:max_context])
2847:
2848:         if source_doc.vector.any() and target_doc.vector.any():
2849:             # Calculate cosine similarity (1 - distance)
2850:             # PERFORMANCE NOTE: Could vectorize this with numpy.dot for batch operations
2851:             similarity = 1 - cosine(source_doc.vector, target_doc.vector)
2852:             return max(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "min_similarity", 0.0), mi
n(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "max_similarity", 1.0), similarity))
2853:
2854:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)
2855:     except Exception:
2856:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_semantic_distance", "default", 0.5)
2857:
2858:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior")
2859:     def _calculate_type_transition_prior(self, source: str, target: str) -> float:
2860:         """Calculate prior based on historical transition frequencies between goal types"""
2861:         source_type = self.nodes[source].type
2862:         target_type = self.nodes[target].type
2863:
2864:         # Define transition probabilities based on logical flow
2865:         # programa → producto → resultado → impacto
2866:         transition_priors = {
2867:             ('programa', 'producto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'producto')", 0.85),
2868:             ('producto', 'resultado'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('producto', 'resultado')", 0.80),
2869:             ('resultado', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('resultado', 'impacto')", 0.75),
2870:             ('programa', 'resultado'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'resultado')", 0.60),
2871:             ('producto', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('producto', 'impacto')", 0.50),
2872:             ('programa', 'impacto'): ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "('programa', 'impacto')", 0.30),
2873:         }
2874:
2875:         # Reverse transitions are less likely
2876:         reverse_key = (target_type, source_type)
2877:         if reverse_key in transition_priors:
2878:             return transition_priors[reverse_key] * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "reverse_multiplier", 0.3)

```

```

2879:
2880:     return transition_priors.get((source_type, target_type), ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior", "default", 0.40))
2881:
2882:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._check_structuralViolation")
2883:     def _check_structuralViolation(self, source: str, target: str) -> str | None:
2884:         """
2885:             AUDIT POINT 2.1: Structural Veto (D6-Q2)
2886:
2887:                 Check if causal link violates structural hierarchy based on TeoriaCambio axioms.
2888:                 Implements set-theoretic constraints per Goertz & Mahoney 2012.
2889:
2890:             Returns:
2891:                 None if link is valid, otherwise a string describing the violation
2892: """
2893:     source_type = self.nodes[source].type
2894:     target_type = self.nodes[target].type
2895:
2896:     # Define causal hierarchy levels (following TeoriaCambio axioms)
2897:     # Lower levels cannot causally influence higher levels
2898:     hierarchy_levels = {
2899:         'programa': 1,
2900:         'producto': 2,
2901:         'resultado': 3,
2902:         'impacto': 4
2903:     }
2904:
2905:     source_level = hierarchy_levels.get(source_type, 0)
2906:     target_level = hierarchy_levels.get(target_type, 0)
2907:
2908:     # Impermissible links: jumping more than 2 levels or reverse causation
2909:     if target_level < source_level:
2910:         # Reverse causation (e.g., Impacto → Producto)
2911:         return f"reverse_causation:{source_type}→{target_type}"
2912:
2913:     if target_level - source_level > 2:
2914:         # Skipping levels (e.g., Programa → Impacto without intermediates)
2915:         return f"level_skip:{source_type}→{target_type} (skips {target_level - source_level - 1} levels)"
2916:
2917:     # Special case: Producto → Impacto is impermissible (must go through Resultado)
2918:     if source_type == 'producto' and target_type == 'impacto':
2919:         return "missing_intermediate:producto→impacto requires resultado"
2920:
2921:     return None
2922:
2923:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity")
2924:     def _calculate_language_specificity(self, keyword: str, policy_area: str | None = None,
2925:                                         context: str | None = None) -> float:
2926:         """Assess specificity of causal language (epistemic certainty)
2927:
2928:             Harmonic Front 3 - Enhancement 4: Language Specificity Assessment
2929:             Enhanced to check policy-specific vocabulary (patrones_verificacion) for current
2930:             Policy Area (P1→P20→P10), not just generic causal keywords.
2931:
2932:             For D6-Q5 (Contextual/Differential Focus): rewards use of specialized terminology
2933:             that anchors intervention in social/cultural context (e.g., "catastro multipropósito",

```

```

2934:     "reparaciÃ³n integral", "mujeres rurales", "guardia indÃ¡-gena").
2935:     """
2936:     # Strong causal indicators
2937:     strong_indicators = ['causa', 'produce', 'genera', 'resulta en', 'conduce a']
2938:     # Moderate indicators
2939:     moderate_indicators = ['permite', 'contribuye', 'facilita', 'mediante', 'a travÃ©s de']
2940:     # Weak indicators
2941:     weak_indicators = ['con el fin de', 'para', 'porque']
2942:
2943:     keyword_lower = keyword.lower()
2944:
2945:     # Base score from causal indicators
2946:     base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "base_score", 0.6) # Refactor
ed
2947:     if any(ind in keyword_lower for ind in strong_indicators):
2948:         base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "strong_indicator_score",
0.9) # Refactored
2949:     elif any(ind in keyword_lower for ind in moderate_indicators):
2950:         base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "moderate_indicator_score",
", 0.7) # Refactored
2951:     elif any(ind in keyword_lower for ind in weak_indicators):
2952:         base_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "weak_indicator_score", 0
.5) # Refactored
2953:
2954:     # HARMONIC FRONT 3 - Enhancement 4: Policy-specific vocabulary boost
2955:     # Check for specialized terminology per policy area
2956:     policy_area_vocabulary = {
2957:         'P1': [ # Ordenamiento Territorial
2958:             'catastro multipropÃ³sito', 'pot', 'pbot', 'eot', 'uaf', 'suelo de protecciÃ³n',
2959:             'zonificaciÃ³n', 'uso del suelo', 'densificaciÃ³n', 'expansÃ³n urbana'
2960:         ],
2961:         'P2': [ # VÃ-ctimas y Paz
2962:             'reparaciÃ³n integral', 'restituciÃ³n de tierras', 'vÃ-ctimas del conflicto',
2963:             'desplazamiento forzado', 'despojo', 'acciÃ³n integral', 'enfoque diferencial Ã©tnico',
2964:             'construcciÃ³n de paz', 'reconciliaciÃ³n', 'memoria histÃ³rica'
2965:         ],
2966:         'P3': [ # Desarrollo Rural
2967:             'mujeres rurales', 'extensiÃ³n agropecuaria', 'asistencia tÃ©cnica rural',
2968:             'adecuaciÃ³n de tierras', 'comercializaciÃ³n campesina', 'economÃ-a campesina',
2969:             'soberanÃ-a alimentaria', 'fondo de tierras'
2970:         ],
2971:         'P4': [ # Grupos Ãltinicos
2972:             'guardia indÃ¡-gena', 'guardia cimarrona', 'territorios colectivos',
2973:             'autoridades ancestrales', 'consulta previa', 'consentimiento libre',
2974:             'medicina tradicional', 'sistema de salud propio indÃ¡-gena', 'jurisdicciÃ³n especial indÃ¡-gena'
2975:         ],
2976:         'P5': [ # Infraestructura y Conectividad
2977:             'terciarias', 'vÃ-as terciarias', 'transporte intermodal', 'Ãºltimo kilÃ³metro',
2978:             'conectividad digital', 'internet rural', 'electrificaciÃ³n rural'
2979:         ],
2980:         'P6': [ # Salud Rural
2981:             'red hospitalaria', 'atenciÃ³n primaria', 'promotores de salud',
2982:             'prevenciÃ³n de enfermedades tropicales', 'saneamiento bÃ¡sico', 'agua segura'
2983:         ],
2984:         'P7': [ # EducaciÃ³n Rural
2985:             'escuela nueva', 'modelos flexibles', 'post-primaria rural',

```

```

2986:         'educaciÃ³n propia', 'alfabetizaciÃ³n', 'deserciÃ³n escolar rural'
2987:     ],
2988:     'P8': [ # Vivienda y Habitabilidad
2989:         'mejoramiento de vivienda rural', 'materiales locales', 'construcciÃ³n sostenible',
2990:         'vivienda de interÃ©s social rural', 'titulaciÃ³n predial'
2991:     ],
2992:     'P9': [ # Medio Ambiente
2993:         'pÃ¡ramos', 'humedales', 'Ã;reas protegidas', 'corredores biolÃ³gicos',
2994:         'servicios ecosistÃ©micos', 'pago por servicios ambientales', 'restauraciÃ³n ecolÃ³gica'
2995:     ],
2996:     'P10': [ # ReactivaciÃ³n EconÃ³mica
2997:         'encadenamientos productivos', 'economÃ-a solidaria', 'cooperativas',
2998:         'microcrÃ©dito', 'emprendimiento asociativo', 'fondo rotatorio'
2999:     ]
3000: }
3001:
3002: # General contextual/differential focus vocabulary (D6-Q5)
3003: contextual_vocabulary = [
3004:     'enfoque diferencial', 'enfoque de gÃ©nero', 'enfoque Ã©tnico',
3005:     'acciÃ³n sin daÃ±o', 'pertinencia cultural', 'contexto territorial',
3006:     'restricciones territoriales', 'barreras culturales', 'inequidad',
3007:     'discriminaciÃ³n', 'exclusiÃ³n', 'vulnerabilidad', 'marginalidad',
3008:     'ruralidad dispersa', 'aislamiento geogrÃ¡fico', 'baja densidad poblacional',
3009:     'poblaciÃ³n dispersa', 'difÃ¢cil acceso'
3010: ]
3011:
3012: # Check for policy-specific vocabulary boost
3013: specificity_boost = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "specificity_boost", 0
.0) # Refactored
3014: text_to_check = (keyword_lower + ' ' + (context or '')).lower()
3015:
3016: if policy_area and policy_area in policy_area_vocabulary:
3017:     for term in policy_area_vocabulary[policy_area]:
3018:         if term.lower() in text_to_check:
3019:             specificity_boost = max(specificity_boost, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_s
pecificity", "policy_specificity_boost", 0.15))
3020:             self.logger.debug(f"Policy-specific term detected: '{term}' for {policy_area}")
3021:             break
3022:
3023: # Check for general contextual vocabulary (D6-Q5)
3024: for term in contextual_vocabulary:
3025:     if term.lower() in text_to_check:
3026:         specificity_boost = max(specificity_boost, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_speci
ficity", "contextual_specificity_boost", 0.10))
3027:         self.logger.debug(f"Contextual term detected: '{term}'")
3028:         break
3029:
3030:     final_score = min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_language_specificity", "final_score_max", 1.0),
base_score + specificity_boost)
3031:
3032:     return final_score
3033:
3034: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence")
3035: def _assess_temporal_coherence(self, source: str, target: str) -> float:
3036:     """Assess temporal coherence based on verb sequences"""
3037:     source_node = self.nodes.get(source)

```

```

3038:     target_node = self.nodes.get(target)
3039:
3040:     if not source_node or not target_node:
3041:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "default", 0.5)
3042:
3043:     # Extract verbs from entity-activity if available
3044:     if source_node.entity_activity and target_node.entity_activity:
3045:         source_verb = source_node.entity_activity.verb_lemma
3046:         target_verb = target_node.entity_activity.verb_lemma
3047:
3048:         # Define logical verb sequences
3049:         verb_sequences = {
3050:             'diagnosticar': 1, 'planificar': 2, 'ejecutar': 3, 'evaluar': 4,
3051:             'diseñar': 2, 'implementar': 3, 'monitorear': 4
3052:         }
3053:
3054:         source_seq = verb_sequences.get(source_verb, 5)
3055:         target_seq = verb_sequences.get(target_verb, 5)
3056:
3057:         if source_seq < target_seq:
3058:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "in_sequence", 0.85)
3059:         elif source_seq == target_seq:
3060:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "same_sequence", 0.60)
3061:         else:
3062:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "reverse_sequence", 0.30)
3063:
3064:     return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_temporal_coherence", "default", 0.50)
3065:
3066: @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency")
3067: def _assess_financial_consistency(self, source: str, target: str) -> float:
3068:     """Assess financial alignment between connected nodes"""
3069:     source_node = self.nodes.get(source)
3070:     target_node = self.nodes.get(target)
3071:
3072:     if not source_node or not target_node:
3073:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "default", 0.5)
3074:
3075:     source_budget = source_node.financial_allocation
3076:     target_budget = target_node.financial_allocation
3077:
3078:     if source_budget and target_budget:
3079:         # Check if budgets are aligned (target should be <= source)
3080:         ratio = target_budget / source_budget if source_budget > 0 else 0
3081:
3082:         if ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_min", 0.1) <= ratio <= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_max", 1.0):
3083:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_score", 0.85)
3084:         elif ratio > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "consistent_max", 1.0) and ratio <= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "slightly_inconsistent_max", 1.5):
3085:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "slightly_inconsistent_score", 0.60)
3086:         else:
3087:             return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "very_inconsistent_score", 0.30)
3088:
3089:     return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._assess_financial_consistency", "default", 0.50)

```

```
3090:
3091:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_textual_proximity")
3092:     def _calculate_textual_proximity(self, source: str, target: str, text: str) -> float:
3093:         """Calculate how often node IDs appear together in text windows"""
3094:         window_size = 200 # characters
3095:         co_occurrences = 0
3096:         total_windows = 0
3097:
3098:         source_positions = [m.start() for m in re.finditer(re.escape(source), text, re.IGNORECASE)]
3099:         target_positions = [m.start() for m in re.finditer(re.escape(target), text, re.IGNORECASE)]
3100:
3101:         for source_pos in source_positions:
3102:             total_windows += 1
3103:             for target_pos in target_positions:
3104:                 if abs(source_pos - target_pos) <= window_size:
3105:                     co_occurrences += 1
3106:                     break
3107:
3108:             if total_windows > 0:
3109:                 proximity_score = co_occurrences / total_windows
3110:             return proximity_score
3111:
3112:     return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_textual_proximity", "default", 0.5)
3113:
3114:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._initialize_prior")
3115:     def _initialize_prior(self, source: str, target: str) -> tuple[float, float, float]:
3116:         """Initialize prior distribution for causal link"""
3117:         # Use type transition as base prior
3118:         type_prior = self._calculate_type_transition_prior(source, target)
3119:
3120:         # Beta distribution parameters - now externalized
3121:         prior_alpha = self.config.get_bayesian_threshold('prior_alpha')
3122:         prior_beta = self.config.get_bayesian_threshold('prior_beta')
3123:
3124:         # Adjust based on type transition
3125:         prior_mean = type_prior
3126:         prior_strength = prior_alpha + prior_beta
3127:
3128:         adjusted_alpha = prior_mean * prior_strength
3129:         adjusted_beta = (1 - prior_mean) * prior_strength
3130:
3131:         return prior_mean, adjusted_alpha, adjusted_beta
3132:
3133:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood")
3134:     def _calculate_composite_likelihood(self, evidence: dict[str, Any]) -> float:
3135:         """Calculate composite likelihood from multiple evidence components
3136:
3137:             Enhanced with:
3138:             - Nonlinear transformation rewarding triangulation
3139:             - Evidence diversity verification across analytical domains
3140:             """
3141:             # Weight different evidence types
3142:             weights = {
3143:                 'semantic_distance': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "semantic_distance_weight", 0.25),
3144:                 'type_transition_prior': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "type_transition_prior_weight", 0.75)
3145:             }
3146:             # Calculate weighted average of evidence components
3147:             weighted_evidence = sum([w * e for w, e in weights.items()])
3148:             return weighted_evidence
3149:
```

```

ion_prior_weight", 0.20),
3145:         'language_specificity': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "language_specificity_weight", 0.20),
3146:         'temporal_coherence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "temporal_coherence_weight", 0.15),
3147:         'financial_consistency': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "financial_consistency_weight", 0.10),
3148:         'textual_proximity': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "textual_proximity_weight", 0.10)
3149:     }
3150:
3151:     # Basic weighted average
3152:     likelihood = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "likelihood", 0.0) # Refactor ed
3153:     evidence_count = 0
3154:     domain_diversity = set()
3155:
3156:     for component, weight in weights.items():
3157:         if component in evidence:
3158:             likelihood += evidence[component] * weight
3159:             evidence_count += 1
3160:
3161:             # Track evidence diversity across domains
3162:             if component in ['semantic_distance', 'textual_proximity']:
3163:                 domain_diversity.add('semantic')
3164:             elif component in ['temporal_coherence']:
3165:                 domain_diversity.add('temporal')
3166:             elif component in ['financial_consistency']:
3167:                 domain_diversity.add('financial')
3168:             elif component in ['type_transition_prior', 'language_specificity']:
3169:                 domain_diversity.add('structural')
3170:
3171:             # Triangulation bonus: Exponentially reward multiple independent observations
3172:             # D6-Q4/Q5 (Adaptiveness/Context) - evidence across different analytical domains
3173:             diversity_count = len(domain_diversity)
3174:             if diversity_count >= 3:
3175:                 # Strong triangulation across semantic, temporal, and financial domains
3176:                 triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_3", 1.15)
3177:             elif diversity_count == 2:
3178:                 # Moderate triangulation
3179:                 triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_2", 1.05)
3180:             else:
3181:                 # Weak or no triangulation
3182:                 triangulation_bonus = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "triangulation_bonus_default", 1.0) # Refactored
3183:
3184:                 # Apply nonlinear transformation
3185:                 enhanced_likelihood = min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "enhanced_likelihood_max", 1.0), likelihood * triangulation_bonus)
3186:
3187:                 # Penalty for insufficient evidence diversity
3188:                 if evidence_count < 3:
3189:                     enhanced_likelihood *= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_composite_likelihood", "insufficient_evidence_penalty", 0.85)

```

```

3190:
3191:         return enhanced_likelihood
3192:
3193:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy")
3194:     def _build_type_hierarchy(self) -> None:
3195:         """Build hierarchy based on goal types"""
3196:
3197:         nodes_by_type: dict[str, list[str]] = defaultdict(list)
3198:         for node_id in self.graph.nodes():
3199:             node_type = self.graph.nodes[node_id].get('type', 'programa')
3200:             nodes_by_type[node_type].append(node_id)
3201:
3202:         # Connect productos to programas
3203:         for prod in nodes_by_type.get('producto', []):
3204:             for prog in nodes_by_type.get('programa', []):
3205:                 if not self.graph.has_edge(prog, prod):
3206:                     self.graph.add_edge(prog, prod, logic='inferido', strength=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy", "inferred_strength", 0.5))
3207:
3208:         # Connect resultados to productos
3209:         for res in nodes_by_type.get('resultado', []):
3210:             for prod in nodes_by_type.get('producto', []):
3211:                 if not self.graph.has_edge(prod, res):
3212:                     self.graph.add_edge(prod, res, logic='inferido', strength=ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._build_type_hierarchy", "inferred_strength", 0.5))
3213:
3214:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence")
3215:     def _calculate_confidence(self, node: MetaNode, link_text: str = "") -> float:
3216:         """
3217:             Calculate confidence score for a causal link.
3218:
3219:             Args:
3220:                 node: The node to calculate confidence for
3221:                 link_text: Optional text describing the causal link
3222:
3223:             Returns:
3224:                 Confidence score between 0 and 1
3225:         """
3226:         confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "confidence", 0.5) # Refactored
3227:
3228:         # Increase confidence if node has quantitative targets
3229:         if node.target and node.baseline:
3230:             try:
3231:                 float(str(node.target).replace(',', '').replace('%', ''))
3232:                 confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "quantitative_target_bonus", 0
.2)
3233:             except (ValueError, TypeError):
3234:                 pass
3235:
3236:         # Increase confidence if text has causal indicators
3237:         if link_text:
3238:             causal_words = ['porque', 'debido', 'mediante', 'a travÃos', 'permite', 'genera', 'produce']
3239:             if any(word in link_text.lower() for word in causal_words):
3240:                 confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "causal_indicator_bonus", 0.15
)
3241:

```

```
3242:     # Increase confidence based on rigor status
3243:     if hasattr(node, 'rigor_status'):
3244:         if node.rigor_status == 'fuerte':
3245:             confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "strong_rigor_bonus", 0.15)
3246:         elif node.rigor_status == 'dÃ©bil':
3247:             confidence -= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "weak_rigor_penalty", 0.1)
3248:
3249:     return min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "max_confidence", 1.0), max(ParameterLoad
erV2.get("farfan_core.analysis.derek_beach.CausalExtractor._calculate_confidence", "min_confidence", 0.0), confidence))
3250:
3251:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._classify_goal_type")
3252:     def _classify_goal_type(self, text: str) -> str:
3253:         """
3254:             Classify the type of a goal based on its text.
3255:
3256:             Args:
3257:                 text: Goal text to classify
3258:
3259:             Returns:
3260:                 Goal type (programa, producto, resultado, impacto)
3261: """
3262:     text_lower = text.lower()
3263:
3264:     # Keywords for each type
3265:     if any(word in text_lower for word in ['programa', 'lÃnea estratÃgica', 'componente', 'eje']):
3266:         return 'programa'
3267:     elif any(word in text_lower for word in ['producto', 'servicio', 'bien', 'actividad']):
3268:         return 'producto'
3269:     elif any(word in text_lower for word in ['resultado', 'efecto', 'cambio', 'mejora']):
3270:         return 'resultado'
3271:     elif any(word in text_lower for word in ['impacto', 'transformaciÃ³n', 'desarrollo', 'bienestar']):
3272:         return 'impacto'
3273:     # Default classification based on position and complexity
3274:     elif len(text) < 100:
3275:         return 'producto'
3276:     else:
3277:         return 'resultado'
3278:
3279:     @calibrated_method("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_justifications")
3280:     def _extract_causal_justifications(self, text: str) -> list[dict[str, Any]]:
3281:         """
3282:             Extract causal justifications from text.
3283:
3284:             Args:
3285:                 text: Text to extract justifications from
3286:
3287:             Returns:
3288:                 List of justifications with text and confidence
3289: """
3290:     justifications = []
3291:
3292:     # Patterns that indicate causal justifications
3293:     patterns = [
3294:         r'porque\s+([^.]+)',
3295:         r'debido\s+a\s+([^.]+)',
3296:         r'mediante\s+([^.]+)',
```

```

3297:         r'a\s+travÃ©s\s+de\s+(^.]+)',
3298:         r'se\s+logra\s+mediante\s+(^.]+)',
3299:         r'permite\s+(^.]+',
3300:         r'genera\s+(^.]+',
3301:     ]
3302:
3303:     for pattern in patterns:
3304:         matches = re.finditer(pattern, text, re.IGNORECASE)
3305:         for match in matches:
3306:             justification_text = match.group(1).strip()
3307:             justifications.append({
3308:                 'text': justification_text,
3309:                 'confidence': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CausalExtractor._extract_causal_justifications", "confidence", 0.7
),
3310:                 'type': 'causal_explanation'
3311:             })
3312:
3313:     return justifications
3314:
3315: class MechanismPartExtractor:
3316:     """Extract Entity-Activity pairs for mechanism parts"""
3317:
3318:     def __init__(self, config: ConfigLoader, nlp_model: spacy.Language) -> None:
3319:         self.logger = logging.getLogger(self.__class__.__name__)
3320:         self.config = config
3321:         self.nlp = nlp_model
3322:         self.entity_aliases = config.get('entity_aliases', {})
3323:
3324:     @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor.extract_entity_activity")
3325:     def extract_entity_activity(self, text: str) -> EntityActivity | None:
3326:         """Extract Entity-Activity tuple from text"""
3327:         doc = self.nlp(text)
3328:
3329:         # Find main verb (activity)
3330:         main_verb = None
3331:         for token in doc:
3332:             if token.pos_ == 'VERB' and token.dep_ in ['ROOT', 'ccomp']:
3333:                 main_verb = token
3334:                 break
3335:
3336:         if not main_verb:
3337:             return None
3338:
3339:         # Find subject entity
3340:         entity = None
3341:         for child in main_verb.children:
3342:             if child.dep_ in ['nsubj', 'nsubjpass']:
3343:                 entity = self._normalize_entity(child.text)
3344:                 break
3345:
3346:         if not entity:
3347:             # Try to find entity from NER
3348:             for ent in doc.ents:
3349:                 if ent.label_ in ['ORG', 'PER']:
3350:                     entity = self._normalize_entity(ent.text)
3351:                     break

```

```

3352:
3353:     if entity and main_verb:
3354:         return EntityActivity(
3355:             entity=entity,
3356:             activity=main_verb.text,
3357:             verb_lemma=main_verb.lemma_,
3358:             confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor.extract_entity_activity", "confidence", 0.85) #
Refactored
3359:         )
3360:
3361:     return None
3362:
3363: @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._normalize_entity")
3364: def _normalize_entity(self, entity: str) -> str:
3365:     """Normalize entity name using aliases"""
3366:     entity_upper = entity.upper().strip()
3367:     return self.entity_aliases.get(entity_upper, entity)
3368:
3369: @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence")
3370: def _calculate_ea_confidence(self, entity: str, activity: str, context: str = "") -> float:
3371:     """
3372:     Calculate confidence for an entity-activity pair.
3373:
3374:     Args:
3375:         entity: Entity text
3376:         activity: Activity text
3377:         context: Surrounding context
3378:
3379:     Returns:
3380:         Confidence score between 0 and 1
3381:     """
3382:     confidence = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "confidence", 0.5) # Refactor
ed
3383:
3384:     # Higher confidence if entity is in known aliases
3385:     if entity.upper() in self.entity_aliases:
3386:         confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "known_entity_bonus", 0.
2)
3387:
3388:     # Higher confidence if activity is a strong verb
3389:     strong_verbs = ['ejecutar', 'implementar', 'desarollar', 'gestionar', 'coordinar']
3390:     if any(verb in activity.lower() for verb in strong_verbs):
3391:         confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "strong_verb_bonus", 0.1
5)
3392:
3393:     # Higher confidence if there's clear grammatical connection in context
3394:     if entity in context and activity in context:
3395:         confidence += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "grammatical_connection_
bonus", 0.15)
3396:
3397:     return min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.MechanismPartExtractor._calculate_ea_confidence", "max_confidence", 1.0), confide
nce)
3398:
3399: @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._find_action_verb")
3400: def _find_action_verb(self, text: str) -> str | None:
3401:     """

```

```
3402:     Find the main action verb in text.
3403:
3404:     Args:
3405:         text: Text to analyze
3406:
3407:     Returns:
3408:         Main action verb or None
3409:     """
3410:     doc = self.nlp(text)
3411:
3412:     # Find main verb
3413:     for token in doc:
3414:         if token.pos_ == 'VERB' and token.dep_ in ['ROOT', 'ccomp', 'xcomp']:
3415:             return token.text
3416:
3417:     # Fallback: any verb
3418:     for token in doc:
3419:         if token.pos_ == 'VERB':
3420:             return token.text
3421:
3422:     return None
3423:
3424: @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._find_subject_entity")
3425: def _find_subject_entity(self, text: str) -> str | None:
3426:     """
3427:     Find the subject entity in text.
3428:
3429:     Args:
3430:         text: Text to analyze
3431:
3432:     Returns:
3433:         Subject entity or None
3434:     """
3435:     doc = self.nlp(text)
3436:
3437:     # Find subject
3438:     for token in doc:
3439:         if token.dep_ in ['nsubj', 'nsubjpass']:
3440:             return self._normalize_entity(token.text)
3441:
3442:     # Try NER
3443:     for ent in doc.ents:
3444:         if ent.label_ in ['ORG', 'PER', 'GPE']:
3445:             return self._normalize_entity(ent.text)
3446:
3447:     return None
3448:
3449: @calibrated_method("farfan_core.analysis.derek_beach.MechanismPartExtractor._validate_entity_activity")
3450: def _validate_entity_activity(self, entity: str, activity: str) -> bool:
3451:     """
3452:     Validate that an entity-activity pair makes sense.
3453:
3454:     Args:
3455:         entity: Entity text
3456:         activity: Activity text
3457:
```

```

3458:     Returns:
3459:         True if valid pair
3460:         """
3461:     # Basic validation
3462:     if not entity or not activity:
3463:         return False
3464:
3465:     # Entity should not be too short or generic
3466:     if len(entity) < 3 or entity.lower() in ['el', 'la', 'los', 'las', 'un', 'una']:
3467:         return False
3468:
3469:     # Activity should be a reasonable verb
3470:     return not len(activity) < 3
3471:
3472: class FinancialAuditor:
3473:     """Financial traceability and auditing"""
3474:
3475:     def __init__(self, config: ConfigLoader) -> None:
3476:         self.logger = logging.getLogger(self.__class__.__name__)
3477:         self.config = config
3478:         self.financial_data: dict[str, dict[str, float]] = {}
3479:         self.unit_costs: dict[str, float] = {}
3480:         self.successful_parses = 0
3481:         self.failed_parses = 0
3482:         self.d3_q3_analysis: dict[str, Any] = {} # Harmonic Front 3 - D3-Q3 metrics
3483:
3484:     @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor.trace_financial_allocation")
3485:     def trace_financial_allocation(self, tables: list[pd.DataFrame],
3486:                                     nodes: dict[str, MetaNode],
3487:                                     graph: nx.DiGraph | None = None) -> dict[str, float]:
3488:         """Trace financial allocations to programs/goals
3489:
3490:             Harmonic Front 3 - Enhancement 5: Single-Case Counterfactual Budget Check
3491:             Incorporates logic from single-case counterfactuals to test minimal sufficiency.
3492:             For D3-Q3 (Traceability/Resources): checks if resource X (BPIN code) were removed,
3493:             would the mechanism (Product) still execute? Only boosts budget traceability score
3494:             if allocation is tied to a specific project.
3495:
3496:             for i, table in enumerate(tables):
3497:                 try:
3498:                     self.logger.info(f"Procesando tabla financiera {i + 1}/{len(tables)}")
3499:                     self._process_financial_table(table, nodes)
3500:                     self.successful_parses += 1
3501:                 except Exception as e:
3502:                     self.logger.error(f"Error procesando tabla financiera {i + 1}: {e}")
3503:                     self.failed_parses += 1
3504:                 continue
3505:
3506:             # HARMONIC FRONT 3 - Enhancement 5: Counterfactual sufficiency check
3507:             if graph is not None:
3508:                 self._perform_counterfactual_budget_check(nodes, graph)
3509:
3510:             self.logger.info(f"Asignaciones financieras trazadas: {len(self.financial_data)}")
3511:             self.logger.info(f"Tablas parseadas exitosamente: {self.successful_parses}, "
3512:                             f"Fallidas: {self.failed_parses}")
3513:             return self.unit_costs

```

```
3514:  
3515:     @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._process_financial_table")  
3516:     def _process_financial_table(self, table: pd.DataFrame,  
3517:                                     nodes: dict[str, MetaNode]) -> None:  
3518:         """Process a single financial table"""  
3519:         # Try to identify relevant columns  
3520:         amount_pattern = re.compile(  
3521:             self.config.get('patterns.financial_headers', r'PRESUPUESTO|VALOR|MONTO'),  
3522:             re.IGNORECASE  
3523:         )  
3524:         program_pattern = re.compile(r'PROGRAMA|META|CÃ223DIGO', re.IGNORECASE)  
3525:  
3526:         amount_col = None  
3527:         program_col = None  
3528:  
3529:         # Search in column names  
3530:         for col in table.columns:  
3531:             col_str = str(col)  
3532:             if amount_pattern.search(col_str) and not amount_col:  
3533:                 amount_col = col  
3534:             if program_pattern.search(col_str) and not program_col:  
3535:                 program_col = col  
3536:  
3537:         # If not found in column names, search in first row  
3538:         if not amount_col or not program_col:  
3539:             first_row = table.iloc[0]  
3540:             for i, val in enumerate(first_row):  
3541:                 val_str = str(val)  
3542:                 if amount_pattern.search(val_str) and not amount_col:  
3543:                     amount_col = i  
3544:                     table.columns = table.iloc[0]  
3545:                     table = table[1:]  
3546:                 if program_pattern.search(val_str) and not program_col:  
3547:                     program_col = i  
3548:                     table.columns = table.iloc[0]  
3549:                     table = table[1:]  
3550:  
3551:         if amount_col is None or program_col is None:  
3552:             self.logger.warning("No se encontraron columnas financieras relevantes")  
3553:             return  
3554:  
3555:         for _, row in table.iterrows():  
3556:             try:  
3557:                 program_id = str(row[program_col]).strip().upper()  
3558:                 amount = self._parse_amount(row[amount_col])  
3559:  
3560:                 if amount and program_id:  
3561:                     matched_node = self._match_program_to_node(program_id, nodes)  
3562:                     if matched_node:  
3563:                         self.financial_data[matched_node] = {  
3564:                             'allocation': amount,  
3565:                             'source': 'budget_table'  
3566:                         }  
3567:  
3568:                         # Update node  
3569:                         nodes[matched_node].financial_allocation = amount
```

```
3570:
3571:             # Calculate unit cost if possible
3572:             node = nodes.get(matched_node)
3573:             if node and node.target:
3574:                 try:
3575:                     target_val = float(str(node.target).replace(',', '').replace('%', ''))
3576:                     if target_val > 0:
3577:                         unit_cost = amount / target_val
3578:                         self.unit_costs[matched_node] = unit_cost
3579:                         nodes[matched_node].unit_cost = unit_cost
3580:                 except (ValueError, TypeError):
3581:                     pass
3582:
3583:             except Exception as e:
3584:                 self.logger.debug(f"Error procesando fila financiera: {e}")
3585:                 continue
3586:
3587: @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._parse_amount")
3588: def _parse_amount(self, value: Any) -> float | None:
3589:     """Parse monetary amount from various formats"""
3590:     if pd.isna(value):
3591:         return None
3592:
3593:     try:
3594:         clean_value = str(value).replace('$', '').replace(',', '').replace(' ', '').replace('.', '')
3595:         # Handle millions/thousands notation
3596:         if 'M' in clean_value.upper() or 'MILLONES' in clean_value.upper():
3597:             clean_value = clean_value.upper().replace('M', '').replace('MILLONES', '')
3598:             return float(clean_value) * 1_000_000
3599:         return float(clean_value)
3600:     except (ValueError, TypeError):
3601:         return None
3602:
3603: @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node")
3604: def _match_program_to_node(self, program_id: str,
3605:                           nodes: dict[str, MetaNode]) -> str | None:
3606:     """Match program ID to existing node using fuzzy matching
3607:
3608:     Enhanced for D1-Q3 / D3-Q3 Financial Traceability:
3609:     - Implements confidence penalty if fuzzy match ratio < 100
3610:     - Reduces node.financial_allocation confidence by 15% for imperfect matches
3611:     - Tracks match quality for overall financial traceability scoring
3612:     """
3613:     if program_id in nodes:
3614:         # Perfect match - no penalty
3615:         return program_id
3616:
3617:     # Try fuzzy matching
3618:     best_match = process.extractOne(
3619:         program_id,
3620:         nodes.keys(),
3621:         scorer=fuzz.ratio,
3622:         score_cutoff=80
3623:     )
3624:
3625:     if best_match:
```

```

3626:     matched_node_id = best_match[0]
3627:     match_ratio = best_match[1]
3628:
3629:     # D1-Q3 / D3-Q3: Apply confidence penalty for non-perfect matches
3630:     if match_ratio < 100:
3631:         penalty_factor = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_to_node", "penalty_factor", 0.85) #
Refactored
3632:     node = nodes[matched_node_id]
3633:
3634:     # Track original allocation before penalty
3635:     if not hasattr(node, '_original_financial_allocation'):
3636:         node._original_financial_allocation = node.financial_allocation
3637:
3638:     # Apply penalty to financial allocation confidence
3639:     if node.financial_allocation:
3640:         penalized_allocation = node.financial_allocation * penalty_factor
3641:         self.logger.debug(
3642:             f"Fuzzy match penalty applied to {matched_node_id}: "
3643:             f"ratio={match_ratio}, penalty={penalty_factor:.2f}, "
3644:             f"allocation {node.financial_allocation:.0f} -> {penalized_allocation:.0f}"
3645:         )
3646:         node.financial_allocation = penalized_allocation
3647:
3648:     # Store match confidence for D1-Q3 / D3-Q3 scoring
3649:     if not hasattr(node, 'financial_match_confidence'):
3650:         node.financial_match_confidence = match_ratio / ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_program_
to_node", "match_confidence_divisor", 100.0)
3651:     else:
3652:         # Average if multiple matches
3653:         node.financial_match_confidence = (node.financial_match_confidence + match_ratio / ParameterLoaderV2.get("farfan_core.analysis.derek_bea
ch.FinancialAuditor._match_program_to_node", "match_confidence_divisor", 100.0)) / 2
3654:
3655:     return matched_node_id
3656:
3657:     return None
3658:
3659: @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check")
3660: def _perform_counterfactual_budget_check(self, nodes: dict[str, MetaNode],
3661:                                         graph: nx.DiGraph) -> None:
3662:     """
3663:     Harmonic Front 3 - Enhancement 5: Counterfactual Sufficiency Test for D3-Q3
3664:
3665:     Tests minimal sufficiency: if resource X (BPIN code) were removed, would the
3666:     mechanism (Product) still execute? Only boosts budget traceability score if
3667:     allocation is tied to a specific project.
3668:
3669:     For D3-Q3 (Traceability/Resources): ensures funding is necessary for the mechanism
3670:     and prevents false positives from generic or disconnected budget entries.
3671:     """
3672:     d3_q3_scores = {}
3673:
3674:     for node_id, node in nodes.items():
3675:         if node.type != 'producto':
3676:             continue
3677:
3678:         # Check if node has financial allocation

```

```

3679:     has_budget = node.financial_allocation is not None and node.financial_allocation > 0
3680:
3681:     # Check if node has entity-activity (mechanism)
3682:     has_mechanism = node.entity_activity is not None
3683:
3684:     # Check if node has dependencies (successors in graph)
3685:     successors = list(graph.successors(node_id)) if graph.has_node(node_id) else []
3686:     has_dependencies = len(successors) > 0
3687:
3688:     # Counterfactual test: Would mechanism still execute without this budget?
3689:     # Check if there are alternative funding sources or generic allocations
3690:     financial_source = self.financial_data.get(node_id, {}).get('source', 'unknown')
3691:     is_specific_allocation = financial_source == 'budget_table' # From specific table entry
3692:
3693:     # Calculate counterfactual necessity score
3694:     # High score = budget is necessary for execution
3695:     # Low score = budget may be generic/disconnected
3696:     necessity_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "necessity_sco
re", 0.0) # Refactored
3697:
3698:     if has_budget and has_mechanism:
3699:         necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "budget_m
echanism_bonus", 0.40) # Budget + mechanism present
3700:
3701:     if has_budget and has_dependencies:
3702:         necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "budget_d
ependency_bonus", 0.30) # Budget supports downstream goals
3703:
3704:     if is_specific_allocation:
3705:         necessity_score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "specific
_allocation_bonus", 0.30) # Specific allocation (not generic)
3706:
3707:     # D3-Q3 quality criteria
3708:     d3_q3_quality = 'insuficiente'
3709:     if necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "excellent
_threshold", 0.85):
3710:         d3_q3_quality = 'excelente'
3711:     elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "good_th
reshold", 0.70):
3712:         d3_q3_quality = 'bueno'
3713:     elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "accepta
ble_threshold", 0.50):
3714:         d3_q3_quality = 'aceptable'
3715:
3716:     d3_q3_scores[node_id] = {
3717:         'necessity_score': necessity_score,
3718:         'd3_q3_quality': d3_q3_quality,
3719:         'has_budget': has_budget,
3720:         'has_mechanism': has_mechanism,
3721:         'has_dependencies': has_dependencies,
3722:         'is_specific_allocation': is_specific_allocation,
3723:         'counterfactual_sufficient': necessity_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfact
ual_budget_check", "sufficient_threshold", 0.50), # Would still execute without budget
3724:         'budget_necessary': necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budg
et_check", "necessary_threshold", 0.70) # Budget is necessary
3725:     }

```

```

3726:
3727:        # Store in node for later retrieval
3728:        node.audit_flags = node.audit_flags or []
3729:        if necessity_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "sufficient_threshold", 0.50):
3730:            node.audit_flags.append('budget_not_necessary')
3731:            self.logger.warning(
3732:                f"D3-Q3: {node_id} may execute without allocated budget (score={necessity_score:.2f})")
3733:            elif necessity_score >= ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check", "excellent_threshold", 0.85):
3734:                node.audit_flags.append('budget_well_traced')
3735:                self.logger.info(f"D3-Q3: {node_id} has well-traced, necessary budget (score={necessity_score:.2f})")
3736:
3737:        # Store aggregate D3-Q3 metrics
3738:        self.d3_q3_analysis = {
3739:            'node_scores': d3_q3_scores,
3740:            'total_products_analyzed': len(d3_q3_scores),
3741:            'well_traced_count': sum(1 for s in d3_q3_scores.values() if s['d3_q3_quality'] == 'excellent'),
3742:            'average_necessity_score': sum(s['necessity_score'] for s in d3_q3_scores.values()) / max(len(d3_q3_scores),
3743:                1)
3744:        }
3745:
3746:        self.logger.info(f"D3-Q3 Counterfactual Budget Check completed: "
3747:                         f"{self.d3_q3_analysis['well_traced_count']}/{len(d3_q3_scores)} "
3748:                         f"products with excellent traceability")
3749:
3750:    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency")
3751:    def _calculate_sufficiency(self, allocation: float, target: float) -> float:
3752:        """
3753:            Calculate if financial allocation is sufficient for target.
3754:
3755:        Args:
3756:            allocation: Financial allocation amount
3757:            target: Target value
3758:
3759:        Returns:
3760:            Sufficiency ratio (ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "auto_param_L2208_31", 1.0)
3761: = exactly sufficient, >ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "auto_param_L2208_58", 1.0) = oversufficient)
3762:        """
3763:        if not target or target == 0:
3764:            return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "default", 0.0)
3765:
3766:        # Calculate unit cost implied by allocation and target
3767:        allocation / target
3768:
3769:        # Compare with historical/expected unit costs if available
3770:        # For now, return simple ratio
3771:        return allocation / target if target > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._calculate_sufficiency", "default", 0.0)
3772:
3773:    @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._detect_allocation_gaps")
3774:    def _detect_allocation_gaps(self, nodes: dict[str, MetaNode]) -> list[dict[str, Any]]:
3775:        """
3776:            Detect gaps in financial allocations.

```

```

3777:     Args:
3778:         nodes: Dictionary of nodes
3779:
3780:     Returns:
3781:         List of detected gaps
3782:     """
3783:     gaps = []
3784:
3785:     for node_id, node in nodes.items():
3786:         # Check for missing allocation
3787:         if node.type in ['producto', 'programa'] and not node.financial_allocation:
3788:             gaps.append({
3789:                 'node_id': node_id,
3790:                 'type': 'missing_allocation',
3791:                 'severity': 'high',
3792:                 'message': f"No financial allocation for {node.type} {node_id}"
3793:             })
3794:
3795:         # Check for insufficient allocation
3796:         if node.financial_allocation and node.target:
3797:             try:
3798:                 target_val = float(str(node.target).replace(',', '').replace('%', ''))
3799:                 if target_val > 0:
3800:                     sufficiency = self._calculate_sufficiency(node.financial_allocation, target_val)
3801:                     if sufficiency < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._detect_allocation_gaps", "sufficiency_threshold", 0.5):
3802:                         gaps.append({
3803:                             'node_id': node_id,
3804:                             'type': 'insufficient_allocation',
3805:                             'severity': 'medium',
3806:                             'message': f"Low sufficiency ratio {sufficiency:.2f} for {node_id}",
3807:                             'sufficiency': sufficiency
3808:                         })
3809:             except (ValueError, TypeError):
3810:                 pass
3811:
3812:     return gaps
3813:
3814:     @calibrated_method("farfan_core.analysis.derek_beach.FinancialAuditor._match_goal_to_budget")
3815:     def _match_goal_to_budget(self, goal_text: str, budget_entries: list[dict[str, Any]]) -> dict[str, Any] | None:
3816:         """
3817:             Match a goal to budget entries.
3818:
3819:         Args:
3820:             goal_text: Goal text to match
3821:             budget_entries: List of budget entries
3822:
3823:         Returns:
3824:             Best matching budget entry or None
3825:         """
3826:         if not budget_entries:
3827:             return None
3828:
3829:         # Extract potential identifiers from goal text
3830:         goal_words = set(goal_text.lower().split())
3831:
```

```
3832:     best_match = None
3833:     best_score = 0
3834:
3835:     for entry in budget_entries:
3836:         entry_text = str(entry.get("description", '')).lower()
3837:         entry_words = set(entry_text.split())
3838:
3839:         # Calculate overlap
3840:         overlap = len(goal_words & entry_words)
3841:         score = overlap / max(len(goal_words), len(entry_words), 1)
3842:
3843:         if score > best_score and score > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.FinancialAuditor._match_goal_to_budget", "score_thresh
old", 0.3): # Minimum threshold
3844:             best_score = score
3845:             best_match = entry
3846:
3847:     return best_match
3848:
3849: class OperationalizationAuditor:
3850:     """Audit operationalization quality"""
3851:
3852:     def __init__(self, config: ConfigLoader) -> None:
3853:         self.logger = logging.getLogger(self.__class__.__name__)
3854:         self.config = config
3855:         self.verb_sequences = config.get('verb_sequences', {})
3856:         self.audit_results: dict[str, AuditResult] = {}
3857:         self.sequence_warnings: list[str] = []
3858:
3859:     @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor.audit_evidence_traceability")
3860:     def audit_evidence_traceability(self, nodes: dict[str, MetaNode]) -> dict[str, AuditResult]:
3861:         """Audit evidence traceability for all nodes
3862:
3863:             Enhanced with D3-Q1 Ficha TÃ©cnica validation:
3864:             - Cross-checks baseline/target against extracted quantitative_claims
3865:             - Verifies DNP INDICATOR_STRUCTURE compliance for producto nodes
3866:             - Scores 'Excelente' only if â\211%80% of productos pass full audit
3867:
3868:             # Import for quantitative claims extraction
3869:             try:
3870:                 from contradiction_deteccion import PolicyContradictionDetectorV2
3871:                 has_detector = True
3872:             except ImportError:
3873:                 has_detector = False
3874:                 self.logger.warning("PolicyContradictionDetectorV2 not available for quantitative claims validation")
3875:
3876:             producto_nodes_count = 0
3877:             producto_nodes_passed = 0
3878:
3879:             for node_id, node in nodes.items():
3880:                 result: AuditResult = {
3881:                     'passed': True,
3882:                     'warnings': [],
3883:                     'errors': [],
3884:                     'recommendations': []
3885:                 }
3886:
```

```

3887:     # Track producto nodes for D3-Q1 scoring
3888:     if node.type == 'producto':
3889:         producto_nodes_count += 1
3890:
3891:     # Extract quantitative claims from node text if detector available
3892:     quantitative_claims = []
3893:     if has_detector:
3894:         try:
3895:             # Create temporary detector instance
3896:             detector = PolicyContradictionDetectorV2(device='cpu')
3897:             quantitative_claims = detector._extract_structured_quantitative_claims(node.text)
3898:         except Exception as e:
3899:             self.logger.debug(f"Could not extract quantitative claims: {e}")
3900:
3901:     # Check baseline
3902:     baseline_valid = False
3903:     if not node.baseline or str(node.baseline).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
3904:         result['errors'].append(f"La base no definida para {node_id}")
3905:         result['passed'] = False
3906:         node.rigor_status = 'd@bil'
3907:         node.audit_flags.append('sin_linea_base')
3908:     else:
3909:         baseline_valid = True
3910:         # Cross-check baseline against quantitative claims (D3-Q1)
3911:         if quantitative_claims:
3912:             baseline_in_claims = any(
3913:                 claim.get('type') in ['indicator', 'target', 'percentage', 'beneficiaries']
3914:                 for claim in quantitative_claims
3915:             )
3916:             if not baseline_in_claims:
3917:                 result['warnings'].append(f"La base no verificada en claims cuantitativos para {node_id}")
3918:
3919:     # Check target
3920:     target_valid = False
3921:     if not node.target or str(node.target).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
3922:         result['errors'].append(f"La meta no definida para {node_id}")
3923:         result['passed'] = False
3924:         node.rigor_status = 'd@bil'
3925:         node.audit_flags.append('sin_meta')
3926:     else:
3927:         target_valid = True
3928:         # Cross-check target against quantitative claims (D3-Q1)
3929:         if quantitative_claims:
3930:             meta_in_claims = any(
3931:                 claim.get('type') == 'target' or 'meta' in claim.get('context', '').lower()
3932:                 for claim in quantitative_claims
3933:             )
3934:             if not meta_in_claims:
3935:                 result['warnings'].append(f"La meta no verificada en claims cuantitativos para {node_id}")
3936:
3937:     # D3-Q1 Ficha T@cnica compliance check for producto nodes
3938:     if node.type == 'producto':
3939:         # Check if has all minimum DNP INDICATOR_STRUCTURE elements
3940:         has_complete_ficha = (
3941:             baseline_valid and
3942:             target_valid and

```



```

3998:     node_data = graph.nodes[node_id]
3999:     if node_data.get('type') == 'programa':
4000:         for successor in graph.successors(node_id):
4001:             if graph.nodes[successor].get('type') == 'producto':
4002:                 programs[node_id].append(successor)
4003:
4004:     # Check sequence within each program
4005:     for program_id, product_goals in programs.items():
4006:         if len(product_goals) < 2:
4007:             continue
4008:
4009:         activities = []
4010:         for goal_id in product_goals:
4011:             node = graph.nodes[goal_id]
4012:             ea = node.get('entity_activity')
4013:             if ea and isinstance(ea, dict):
4014:                 verb = ea.get('verb_lemma', '')
4015:                 sequence_num = self.verb_sequences.get(verb, 999)
4016:                 activities.append((goal_id, verb, sequence_num))
4017:
4018:     # Check for sequence violations
4019:     activities.sort(key=lambda x: x[2])
4020:     for i in range(len(activities) - 1):
4021:         if activities[i][2] > activities[i + 1][2]:
4022:             warning = (f"ViolaciÃ³n de secuencia en {program_id}: "
4023:                        f"{activities[i][1]} ({activities[i][0]}) "
4024:                        f"antes de {activities[i + 1][1]} ({activities[i + 1][0]})")
4025:             warnings.append(warning)
4026:             self.logger.warning(warning)
4027:
4028:     self.sequence_warnings = warnings
4029:     return warnings
4030:
4031: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit")
4032: def bayesian_counterfactual_audit(self, nodes: dict[str, MetaNode],
4033:                                     graph: nx.DiGraph,
4034:                                     historical_data: dict[str, Any] | None = None,
4035:                                     pdet_alignment: float | None = None) -> dict[str, Any]:
4036:     """
4037:     AGUJA III: El Auditor Contrafactual Bayesiano
4038:     Perform counterfactual audit using Bayesian causal reasoning
4039:
4040:     Harmonic Front 3: Enhanced to consume pdet_alignment scores for D4-Q5 and D5-Q4 integration
4041:     """
4042:     self.logger.info("Iniciando auditorÃ-a contrafactual Bayesiana...")
4043:
4044:     # Build implicit Structural Causal Model (SCM)
4045:     scm_dag = self._build_normative_dag()
4046:
4047:     # Initialize historical priors
4048:     if historical_data is None:
4049:         historical_data = self._get_default_historical_priors()
4050:
4051:     # Audit results by layers
4052:     layer1_results = self._audit_direct_evidence(nodes, scm_dag, historical_data)
4053:     layer2_results = self._audit_causal_implications(nodes, graph, layer1_results)

```

```

4054:     layer3_results = self._audit_systemic_risk(nodes, graph, layer1_results, layer2_results, pdet_alignment)
4055:
4056:     # Generate optimal remediation recommendations
4057:     recommendations = self._generate_optimal_remediations(
4058:         layer1_results, layer2_results, layer3_results
4059:     )
4060:
4061:     audit_report = {
4062:         'direct_evidence': layer1_results,
4063:         'causal_implications': layer2_results,
4064:         'systemic_risk': layer3_results,
4065:         'recommendations': recommendations,
4066:         'summary': {
4067:             'total_nodes': len(nodes),
4068:             'critical_omissions': sum(1 for r in layer1_results.values()
4069:                                       if r.get('omission_severity') == 'critical'),
4070:             'expected_success_probability': layer3_results.get('success_probability', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit", "success_probability_default", 0.0)),
4071:             'risk_score': layer3_results.get('risk_score', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit", "risk_score_default", 0.0))
4072:         }
4073:     }
4074:
4075:     self.logger.info(f"Auditora-a contrafactual completada: "
4076:                      f"{audit_report['summary']['critical_omissions']} omisiones críticas detectadas")
4077:
4078:     return audit_report
4079:
4080: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._build_normative_dag")
4081: def _build_normative_dag(self) -> nx.DiGraph:
4082:     """Build normative DAG of expected relationships in well-formed plans"""
4083:     dag = nx.DiGraph()
4084:
4085:     # Define normative structure
4086:     # Each goal type should have these attributes
4087:     dag.add_node('baseline', type='required_attribute')
4088:     dag.add_node('target', type='required_attribute')
4089:     dag.add_node('entity', type='required_attribute')
4090:     dag.add_node('budget', type='recommended_attribute')
4091:     dag.add_node('mechanism', type='recommended_attribute')
4092:     dag.add_node('timeline', type='optional_attribute')
4093:     dag.add_node('risk_factors', type='optional_attribute')
4094:
4095:     # Causal relationships
4096:     dag.add_edge('baseline', 'target', relation='defines_gap')
4097:     dag.add_edge('entity', 'mechanism', relation='executes')
4098:     dag.add_edge('budget', 'mechanism', relation='enables')
4099:     dag.add_edge('mechanism', 'target', relation='achieves')
4100:     dag.add_edge('risk_factors', 'target', relation='threatens')
4101:
4102:     return dag
4103:
4104: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors")
4105: def _get_default_historical_priors(self) -> dict[str, Any]:
4106:     """Get default historical priors if no data is available"""
4107:     return {

```

```

4108:         'entity_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4109:                         "entity_presence_success_rate", 0.94),
4110:         'baseline_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4111:                         "baseline_presence_success_rate", 0.89),
4112:         'target_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4113:                         "target_presence_success_rate", 0.92),
4114:         'budget_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4115:                         "budget_presence_success_rate", 0.78),
4116:         'mechanism_presence_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4117:                         "mechanism_presence_success_rate", 0.65),
4118:         'complete_documentation_success_rate': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors",
4119:                         "complete_documentation_success_rate", 0.82),
4120:         'node_type_success_rates': {
4121:             'producto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "producto_suc-
cess_rate", 0.85),
4122:             'resultado': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "resultado_s-
uccess_rate", 0.72),
4123:             'impacto': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_default_historical_priors", "impacto_succe-
ss_rate", 0.58)
4124:         }
4125:     }
4126:
4127:     @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence")
4128:     def _audit_direct_evidence(self, nodes: dict[str, MetaNode],
4129:                               scm_dag: nx.DiGraph,
4130:                               historical_data: dict[str, Any]) -> dict[str, dict[str, Any]]:
4131:         """Layer 1: Audit direct evidence of required components
4132: Enhanced with highly specific Bayesian priors for rare evidence items.
4133: Example: D2-Q4 risk matrix, D5-Q5 unwanted effects are rare in poor PDMs.
4134:
4135: results = {}
4136:
4137: # Load highly specific priors for rare evidence types
4138: # D2-Q4: Risk matrices are rare in poor PDMs (high probative value as Smoking Gun)
4139: rare_evidence_priors = {
4140:     'risk_matrix': {
4141:         'prior_alpha': 1.5, # Low alpha = rare occurrence
4142:         'prior_beta': 12.0, # High beta = high failure rate when absent
4143:         'keywords': ['matriz de riesgo', 'análisis de riesgo', 'gestión de riesgo', 'riesgos identificados']
4144:     },
4145:     'unwanted_effects': {
4146:         'prior_alpha': 1.8, # D5-Q5: Effects analysis is also rare
4147:         'prior_beta': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "unwanted_effects_p-
rior_beta", 10.5),
4148:         'keywords': ['efectos no deseados', 'efectos adversos', 'impactos negativos',
4149:                     'consecuencias no previstas']
4150:     },
4151:     'theory_of_change': {
4152:         'prior_alpha': 1.2,
4153:         'prior_beta': 15.0,
4154:         'keywords': ['teoría de cambio', 'teoría del cambio', 'cadena causal', 'modelo lógico']
4155:     }
4156:
4157:     for node_id, node in nodes.items():

```

```

4154:     omissions = []
4155:     omission_probs = {}
4156:     rare_evidence_found = {}
4157:
4158:     # Check for rare, high-value evidence in node text
4159:     node_text_lower = node.text.lower()
4160:     for evidence_type, prior_config in rare_evidence_priors.items():
4161:         if any(kw in node_text_lower for kw in prior_config['keywords']):
4162:             # Rare evidence found! Strong Smoking Gun
4163:             rare_evidence_found[evidence_type] = {
4164:                 'prior_alpha': prior_config['prior_alpha'],
4165:                 'prior_beta': prior_config['prior_beta'],
4166:                 'posterior_strength': prior_config['prior_alpha'] / (
4167:                     prior_config['prior_alpha'] + prior_config['prior_beta'])
4168:             }
4169:             self.logger.info(f"Rare evidence '{evidence_type}' found in {node_id} - Strong Smoking Gun!")
4170:
4171:     # Check baseline
4172:     if not node.baseline or str(node.baseline).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
4173:         p_failure_given_omission = 1.0 - historical_data.get('baseline_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "baseline_presence_success_rate", 0.89))
4174:         omissions.append('baseline')
4175:         omission_probs['baseline'] = p_failure_given_omission
4176:
4177:     # Check target
4178:     if not node.target or str(node.target).upper() in ['ND', 'POR DEFINIR', 'N/A', 'NONE']:
4179:         p_failure_given_omission = 1.0 - historical_data.get('target_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "target_presence_success_rate", 0.92))
4180:         omissions.append('target')
4181:         omission_probs['target'] = p_failure_given_omission
4182:
4183:     # Check entity
4184:     if not node.responsible_entity:
4185:         p_failure_given_omission = 1.0 - historical_data.get('entity_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "entity_presence_success_rate", 0.94))
4186:         omissions.append('entity')
4187:         omission_probs['entity'] = p_failure_given_omission
4188:
4189:     # Check budget
4190:     if not node.financial_allocation:
4191:         p_failure_given_omission = 1.0 - historical_data.get('budget_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "budget_presence_success_rate", 0.78))
4192:         omissions.append('budget')
4193:         omission_probs['budget'] = p_failure_given_omission
4194:
4195:     # Check mechanism
4196:     if not node.entity_activity:
4197:         p_failure_given_omission = 1.0 - historical_data.get('mechanism_presence_success_rate', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "mechanism_presence_success_rate", 0.65))
4198:         omissions.append('mechanism')
4199:         omission_probs['mechanism'] = p_failure_given_omission
4200:
4201:     # Determine severity
4202:     severity = 'none'
4203:     if omission_probs:
4204:         max_failure_prob = max(omission_probs.values())

```

```

4205:             if max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "critical_threshold", 0.15):
4206:                 severity = 'critical'
4207:             elif max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "high_threshold", 0.10):
4208:                 severity = 'high'
4209:             elif max_failure_prob > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence", "medium_threshold", 0.05):
4210:                 severity = 'medium'
4211:             else:
4212:                 severity = 'low'
4213:
4214:             results[node_id] = {
4215:                 'omissions': omissions,
4216:                 'omission_probabilities': omission_probs,
4217:                 'omission_severity': severity,
4218:                 'node_type': node.type,
4219:                 'rare_evidence_found': rare_evidence_found # Add rare evidence to results
4220:             }
4221:
4222:         return results
4223:
4224:     @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications")
4225:     def _audit_causal_implications(self, nodes: dict[str, MetaNode],
4226:                                     graph: nx.DiGraph,
4227:                                     direct_evidence: dict[str, dict[str, Any]]) -> dict[str, dict[str, Any]]:
4228:         """Layer 2: Audit causal implications of omissions"""
4229:         implications = {}
4230:
4231:         for node_id, node in nodes.items():
4232:             node_omissions = direct_evidence[node_id]['omissions']
4233:             causal_effects = {}
4234:
4235:             # If baseline is missing
4236:             if 'baseline' in node_omissions:
4237:                 # P(target_miscalibrated | missing_baseline)
4238:                 causal_effects['target_miscalibration'] = {
4239:                     'probability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "target_miscalibration_prob", 0.73),
4240:                     'description': 'Sin lÃnea base, la meta probablemente estÃ¡ mal calibrada'
4241:                 }
4242:
4243:             # If entity and high budget are missing
4244:             if 'entity' in node_omissions and node.financial_allocation and node.financial_allocation > 1000000:
4245:                 causal_effects['implementation_failure'] = {
4246:                     'probability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "implementation_failure_high_budget_prob", 0.89),
4247:                     'description': 'Alto presupuesto sin entidad responsable indica alto riesgo de falla'
4248:                 }
4249:             elif 'entity' in node_omissions:
4250:                 causal_effects['implementation_failure'] = {
4251:                     'probability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "implementation_failure_prob", 0.65),
4252:                     'description': 'Sin entidad responsable, la implementaciÃ³n es incierta'
4253:                 }
4254:

```

```

4255:         # If mechanism is missing
4256:         if 'mechanism' in node_omissions:
4257:             causal_effects['unclear_pathway'] = {
4258:                 'probability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "unclear_p
athway_prob", 0.70),
4259:                 'description': 'Sin mecanismo definido, la vÃ-a causal es opaca'
4260:             }
4261:
4262:         # Check downstream effects
4263:         successors = list(graph.successors(node_id)) if graph.has_node(node_id) else []
4264:         if node_omissions and successors:
4265:             causal_effects['cascade_risk'] = {
4266:                 'probability': min(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "casca
de_risk_max_prob", 0.95), ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "cascade_risk_base_prob",
0.4) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications", "cascade_risk_per_omission_prob", 0.1) * len(
node_omissions)),
4267:                 'affected_nodes': successors,
4268:                 'description': f'Omisiones pueden afectar {len(successors)} nodos dependientes'
4269:             }
4270:
4271:         implications[node_id] = {
4272:             'causal_effects': causal_effects,
4273:             'total_risk': sum(e['probability'] for e in causal_effects.values()) / max(len(causal_effects), 1)
4274:         }
4275:
4276:     return implications
4277:
4278: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk")
4279: def _audit_systemic_risk(self, nodes: dict[str, MetaNode],
4280:                         graph: nx.DiGraph,
4281:                         direct_evidence: dict[str, dict[str, Any]],
4282:                         causal_implications: dict[str, dict[str, Any]],
4283:                         pdet_alignment: float | None = None) -> dict[str, Any]:
4284:
4285:     """
4286:     AUDIT POINT 2.3: Policy Alignment Dual Constraint
4287:     Layer 3: Calculate systemic risk from accumulated omissions
4288:
4289:     Harmonic Front 3 - Enhancement 1: Alignment and Systemic Risk Linkage
4290:     Incorporates Policy Alignment scores (PND, ODS, RRI) as variable in systemic risk.
4291:
4292:     For D5-Q4 (Riesgos SistÃ@micos) and D4-Q5 (AlineaciÃ³n):
4293:     - If pdet_alignment >= 0.60, applies 1.2/227 multiplier to risk_score
4294:     - Excelente on D5-Q4 requires risk_score < 0.10)
4295:
4296:     Implements dual constraints integrating macro-micro causality per Lieberman 2015.
4297:
4298:     # Identify critical nodes (high centrality)
4299:     if graph.number_of_nodes() > 0:
4300:         try:
4301:             centrality = nx.betweenness_centrality(graph)
4302:         except (nx.NetworkXError, ZeroDivisionError, Exception) as e:
4303:             logging.warning(f"Failed to calculate betweenness centrality: {e}. Using default values.")
4304:             centrality = dict.fromkeys(graph.nodes(), 0.5)
4305:
4306:     else:
4307:         centrality = {}

```

```

4307:
4308:     # Calculate P(cascade_failure | omission_set)
4309:     critical_omissions = []
4310:     for node_id, evidence in direct_evidence.items():
4311:         if evidence['omission_severity'] in ['critical', 'high']:
4312:             node_centrality = centrality.get(node_id, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "default_centrality", 0.5))
4313:             critical_omissions.append({
4314:                 'node_id': node_id,
4315:                 'severity': evidence['omission_severity'],
4316:                 'centrality': node_centrality,
4317:                 'omissions': evidence['omissions']
4318:             })
4319:
4320:     # Calculate systemic risk
4321:     if critical_omissions:
4322:         # Weighted by centrality
4323:         risk_score = sum(
4324:             (ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "critical_severity_multiplier", 1.0) if om['severity'] == 'critical' else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "high_severity_multiplier", 0.7)) * (om['centrality'] + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "centrality_bonus", 0.1))
4325:                 for om in critical_omissions
4326:             ) / len(nodes)
4327:     else:
4328:         risk_score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_score", 0.0) # Refactored
4329:
4330:     # AUDIT POINT 2.3: Policy Alignment Dual Constraint
4331:     # If pdet_alignment != None ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "alignment_threshold", 0.60), apply 1.2*227 multiplier to risk_score
4332:     # This enforces integration between D4-Q5 (AlineaciÃ³n) and D5-Q4 (Riesgos SistÃ©micos)
4333:     alignment_penalty_applied = False
4334:     alignment_threshold = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "alignment_threshold", 0.6) # Refactored
4335:     alignment_multiplier = 1.2
4336:
4337:     if pdet_alignment is not None and pdet_alignment <= alignment_threshold:
4338:         original_risk = risk_score
4339:         risk_score = risk_score * alignment_multiplier
4340:         alignment_penalty_applied = True
4341:         self.logger.warning(
4342:             f"ALIGNMENT PENALTY (D5-Q4): pdet_alignment={pdet_alignment:.2f} > alignment_threshold, "
4343:             f"risk_score escalated from {original_risk:.3f} to {risk_score:.3f} "
4344:             f"(multiplier: {alignment_multiplier}*227). Dual constraint per Lieberman 2015."
4345:         )
4346:
4347:     # Calculate P(success | current_state)
4348:     total_omissions = sum(len(e['omissions']) for e in direct_evidence.values())
4349:     total_possible = len(nodes) * 5 # 5 key attributes per node
4350:     completeness = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "completeness_factor", 1.0) - (total_omissions / max(total_possible, 1))
4351:
4352:     # Success probability (simplified Bayesian)
4353:     base_success_rate = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "base_success_rate", 0.7) # Refactored

```

```

4354:     success_probability = base_success_rate * completeness
4355:
4356:     # D5-Q4 quality criteria check (AUDIT POINT 2.3)
4357:     # Excellent requires risk_score < ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_excellent", 0.10) (matching ODS benchmarks per UN 2020)
4358:     d5_q4_quality = 'insuficiente'
4359:     risk_threshold_excellent = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_excellent", 0.1) # Refactored
4360:     risk_threshold_good = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_good", 0.2) # Refactored
4361:     risk_threshold_acceptable = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk", "risk_threshold_acceptable", 0.35) # Refactored
4362:
4363:     if risk_score < risk_threshold_excellent:
4364:         d5_q4_quality = 'excelente'
4365:     elif risk_score < risk_threshold_good:
4366:         d5_q4_quality = 'bueno'
4367:     elif risk_score < risk_threshold_acceptable:
4368:         d5_q4_quality = 'aceptable'
4369:
4370:     # Flag if alignment is causing quality failure
4371:     alignment_causing_failure = (
4372:         alignment_penalty_applied and
4373:         original_risk < risk_threshold_excellent and
4374:         risk_score >= risk_threshold_excellent
4375:     )
4376:
4377:     return {
4378:         'risk_score': min(1.0, risk_score),
4379:         'success_probability': success_probability,
4380:         'critical_omissions': critical_omissions,
4381:         'completeness': completeness,
4382:         'total_omissions': total_omissions,
4383:         'pdet_alignment': pdet_alignment,
4384:         'alignment_penalty_applied': alignment_penalty_applied,
4385:         'alignment_threshold': alignment_threshold,
4386:         'alignment_multiplier': alignment_multiplier,
4387:         'alignment_causing_failure': alignment_causing_failure,
4388:         'd5_q4_quality': d5_q4_quality,
4389:         'd4_q5_alignment_score': pdet_alignment,
4390:         'risk_thresholds': {
4391:             'excellent': risk_threshold_excellent,
4392:             'good': risk_threshold_good,
4393:             'acceptable': risk_threshold_acceptable
4394:         }
4395:     }
4396:
4397:     @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._generate_optimal_remediations")
4398:     def _generate_optimal_remediations(self,
4399:                                         direct_evidence: dict[str, dict[str, Any]],
4400:                                         causal_implications: dict[str, dict[str, Any]],
4401:                                         systemic_risk: dict[str, Any]) -> list[dict[str, Any]]:
4402:         """Generate prioritized remediation recommendations"""
4403:         remediations = []
4404:
4405:         # Calculate expected value of information for each remediation

```

```

4406:     for node_id, evidence in direct_evidence.items():
4407:         if not evidence['omissions']:
4408:             continue
4409:
4410:         for omission in evidence['omissions']:
4411:             # Estimate impact
4412:             omission_prob = evidence['omission_probabilities'].get(omission, ParameterLoaderV2.get("farfan_core.analysis.derek_beach.OperationalizationA
uditor._generate_optimal_remediations", "default_omission_prob", 0.1))
4413:             causal_risk = causal_implications[node_id]['total_risk']
4414:
4415:             # Expected value = P(failure_avoided) * Impact
4416:             expected_value = omission_prob * (1 + causal_risk)
4417:
4418:             # Effort estimate (simplified)
4419:             effort_map = {
4420:                 'baseline': 3, # Moderate effort to research
4421:                 'target': 2, # Low effort to define
4422:                 'entity': 2, # Low effort to assign
4423:                 'budget': 4, # Higher effort to allocate
4424:                 'mechanism': 5 # Highest effort to design
4425:             }
4426:             effort = effort_map.get(omission, 3)
4427:
4428:             # Priority = Expected Value / Effort
4429:             priority = expected_value / effort
4430:
4431:             remediations.append({
4432:                 'node_id': node_id,
4433:                 'omission': omission,
4434:                 'severity': evidence['omission_severity'],
4435:                 'expected_value': expected_value,
4436:                 'effort': effort,
4437:                 'priority': priority,
4438:                 'recommendation': self._get_remediation_text(omission, node_id)
4439:             })
4440:
4441:             # Sort by priority (descending)
4442:             remediations.sort(key=lambda x: x['priority'], reverse=True)
4443:
4444:             return remediations
4445:
4446: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._get_remediation_text")
4447: def _get_remediation_text(self, omission: str, node_id: str) -> str:
4448:     """Get specific remediation text for an omission"""
4449:     texts = {
4450:         'baseline': f"Definir lÃnea base cuantitativa para {node_id} basada en diagnÃstico actual",
4451:         'target': f"Especificar meta cuantitativa alcanzable para {node_id} con horizonte temporal",
4452:         'entity': f"Asignar entidad responsable clara para la ejecuciÃ³n de {node_id}",
4453:         'budget': f"Asignar recursos presupuestarios especÃficos a {node_id}",
4454:         'mechanism': f"Documentar mecanismo causal (Entidad-Actividad) para {node_id}"
4455:     }
4456:     return texts.get(omission, f"Completar {omission} para {node_id}")
4457:
4458: @calibrated_method("farfan_core.analysis.derek_beach.OperationalizationAuditor._perform_counterfactual_budget_check")
4459: def _perform_counterfactual_budget_check(self, nodes: dict[str, MetaNode],
4460:                                         graph: nx.DiGraph) -> dict[str, Any]:

```

```

4461:     """
4462:     Perform counterfactual budget check for operationalization audit.
4463:
4464:     This method evaluates whether removing budget allocation would prevent
4465:     goal execution, helping identify necessary vs. superfluous allocations.
4466:
4467:     Args:
4468:         nodes: Dictionary of meta nodes
4469:         graph: Causal graph
4470:
4471:     Returns:
4472:         Dictionary with counterfactual analysis results
4473:     """
4474:     results = {
4475:         'nodes_analyzed': 0,
4476:         'budget_necessary': [],
4477:         'budget_optional': [],
4478:         'unallocated': []
4479:     }
4480:
4481:     for node_id, node in nodes.items():
4482:         results['nodes_analyzed'] += 1
4483:
4484:         has_budget = node.financial_allocation and node.financial_allocation > 0
4485:         has_mechanism = node.entity_activity is not None
4486:         has_dependencies = len(list(graph.successors(node_id))) > 0 if graph.has_node(node_id) else False
4487:
4488:         if not has_budget:
4489:             results['unallocated'].append(node_id)
4490:         elif has_mechanism and has_dependencies:
4491:             # Budget seems necessary for execution
4492:             results['budget_necessary'].append(node_id)
4493:         else:
4494:             # Budget may be optional or disconnected
4495:             results['budget_optional'].append(node_id)
4496:
4497:     return results
4498:
4499: class BayesianMechanismInference:
4500:     """
4501:     AGUJA II: El Modelo Generativo de Mecanismos
4502:     Hierarchical Bayesian model for causal mechanism inference
4503:
4504:     F1.2 ARCHITECTURAL REFACTORING:
4505:     This class now integrates with refactored Bayesian engine components:
4506:     - BayesianPriorBuilder: Construye priors adaptativos (AGUJA I)
4507:     - BayesianSamplingEngine: Ejecuta MCMC sampling (AGUJA II)
4508:     - NecessitySufficiencyTester: Ejecuta Hoop Tests (AGUJA III)
4509:
4510:     The refactored components provide:
4511:     - Crystal-clear separation of concerns
4512:     - Trivial unit testing
4513:     - Explicit compliance with Fronts B and C
4514:
4515:     Legacy methods are preserved for backward compatibility.
4516:     """

```

```
4517:  
4518:     def __init__(self, config: ConfigLoader, nlp_model: spacy.Language, **kwargs) -> None:  
4519:         """  
4520:             Initialize Bayesian Mechanism Inference engine.  
4521:  
4522:             Args:  
4523:                 config: Configuration loader instance  
4524:                 nlp_model: spaCy NLP model for text processing  
4525:                 **kwargs: Accepts additional keyword arguments for backward compatibility.  
4526:                     Unexpected arguments (e.g., 'causal_hierarchy') are logged and ignored.  
4527:  
4528:             Note:  
4529:                 This function signature has been made defensive to handle unexpected  
4530:                 keyword arguments that may be passed due to interface drift.  
4531:             """  
4532:             # Log warning if unexpected kwargs are passed  
4533:             if kwargs:  
4534:                 logging.getLogger(__name__).warning(  
4535:                     f"BayesianMechanismInference.__init__ received unexpected keyword arguments: {list(kwargs.keys())}. "  
4536:                     "These will be ignored. Expected signature: __init__(self, config: ConfigLoader, nlp_model: spacy.Language)"  
4537:                 )  
4538:  
4539:             self.logger = logging.getLogger(self.__class__.__name__)  
4540:             self.config = config  
4541:             self.nlp = nlp_model  
4542:  
4543:             # F1.2: Initialize refactored Bayesian engine adapter if available  
4544:             if REFACTORED_BAYESIAN_AVAILABLE:  
4545:                 try:  
4546:                     self.bayesian_adapter = BayesianEngineAdapter(config, nlp_model)  
4547:                     if self.bayesian_adapter.is_available():  
4548:                         self.logger.info("â\234\223 Usando motor Bayesiano refactorizado (F1.2)")  
4549:                         self._log_refactored_components()  
4550:                 else:  
4551:                     self.bayesian_adapter = None  
4552:             except Exception as e:  
4553:                 self.logger.warning(f"Error inicializando motor refactorizado: {e}")  
4554:             self.bayesian_adapter = None  
4555:         else:  
4556:             self.bayesian_adapter = None  
4557:  
4558:         # Load mechanism type hyperpriors from configuration (externalized)  
4559:         self.mechanism_type_priors = {  
4560:             'administrativo': self.config.get_mechanism_prior('administrativo'),  
4561:             'tecnico': self.config.get_mechanism_prior('tecnico'),  
4562:             'financiero': self.config.get_mechanism_prior('financiero'),  
4563:             'politico': self.config.get_mechanism_prior('politico'),  
4564:             'mixto': self.config.get_mechanism_prior('mixto')  
4565:         }  
4566:  
4567:         # Typical activity sequences by mechanism type  
4568:         # These could also be externalized if needed for domain-specific customization  
4569:         self.mechanism_sequences = {  
4570:             'administrativo': ['planificar', 'coordinar', 'gestionar', 'supervisar'],  
4571:             'tecnico': ['diagnosticar', 'diseñar', 'implementar', 'evaluar'],  
4572:             'financiero': ['asignar', 'ejecutar', 'auditar', 'reportar'],
```

```

4573:         'politico': ['concertar', 'negociar', 'aprobar', 'promulgar']
4574:     }
4575:
4576:     # Track inferred mechanisms
4577:     self.inferred_mechanisms: dict[str, dict[str, Any]] = {}
4578:
4579: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._log_refactored_components")
4580: def _log_refactored_components(self) -> None:
4581:     """Log status of refactored Bayesian components (F1.2)"""
4582:     if self.bayesian_adapter:
4583:         status = self.bayesian_adapter.get_component_status()
4584:         self.logger.info(" - BayesianPriorBuilder: " +
4585:                           ("â\234\223" if status['prior_builder_ready'] else "â\234\227"))
4586:         self.logger.info(" - BayesianSamplingEngine: " +
4587:                           ("â\234\223" if status['sampling_engine_ready'] else "â\234\227"))
4588:         self.logger.info(" - NecessitySufficiencyTester: " +
4589:                           ("â\234\223" if status['necessity_tester_ready'] else "â\234\227"))
4590:
4591: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms")
4592: def infer_mechanisms(self, nodes: dict[str, MetaNode],
4593:                      text: str) -> dict[str, dict[str, Any]]:
4594: """
4595: Infer latent causal mechanisms using hierarchical Bayesian modeling
4596:
4597: HARMONIC FRONT 4 ENHANCEMENT:
4598: - Tracks mean mechanism_type uncertainty for quality criteria
4599: - Reports uncertainty reduction metrics
4600: """
4601: self.logger.info("Iniciando inferencia Bayesiana de mecanismos...")
4602:
4603: # Focus on 'producto' nodes which should have mechanisms
4604: product_nodes = {nid: n for nid, n in nodes.items() if n.type == 'producto'}
4605:
4606: # Track uncertainties for mean calculation
4607: mechanism_uncertainties = []
4608:
4609: for node_id, node in product_nodes.items():
4610:     mechanism = self._infer_single_mechanism(node, text, nodes)
4611:     self.inferred_mechanisms[node_id] = mechanism
4612:
4613:     # Track mechanism type uncertainty for quality criteria
4614:     if 'uncertainty' in mechanism:
4615:         mech_type_uncertainty = mechanism['uncertainty'].get('mechanism_type', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms", "default_uncertainty", 1.0))
4616:         mechanism_uncertainties.append(mech_type_uncertainty)
4617:
4618:     # Calculate mean mechanism uncertainty for Harmonic Front 4 quality criteria
4619:     mean_mech_uncertainty = (
4620:         np.mean(mechanism_uncertainties) if mechanism_uncertainties else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms", "default_mean_uncertainty", 1.0)
4621:     )
4622:
4623:     self.logger.info(f"Mechanismos inferidos: {len(self.inferred_mechanisms)}")
4624:     self.logger.info(f"Mean mechanism_type uncertainty: {mean_mech_uncertainty:.4f}")
4625:
4626:     # Store for reporting

```

```
4627:         self._mean_mechanism_uncertainty = mean_mech_uncertainty
4628:
4629:     return self.inferred_mechanisms
4630:
4631: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_single_mechanism")
4632: def _infer_single_mechanism(self, node: MetaNode, text: str,
4633:                             all_nodes: dict[str, MetaNode]) -> dict[str, Any]:
4634:     """Infer mechanism for a single product node"""
4635:     # Extract observations from text
4636:     observations = self._extract_observations(node, text)
4637:
4638:     # Level 3: Sample mechanism type from hyperprior
4639:     mechanism_type_posterior = self._infer_mechanism_type(observations)
4640:
4641:     # Level 2: Infer activity sequence given mechanism type
4642:     sequence_posterior = self._infer_activity_sequence(
4643:         observations, mechanism_type_posterior
4644:     )
4645:
4646:     # Level 1: Calculate coherence factor
4647:     coherence_score = self._calculate_coherence_factor(
4648:         node, observations, all_nodes
4649:     )
4650:
4651:     # Validation tests
4652:     sufficiency = self._test_sufficiency(node, observations)
4653:     necessity = self._test_necessity(node, observations)
4654:
4655:     # Quantify uncertainty
4656:     uncertainty = self._quantify_uncertainty(
4657:         mechanism_type_posterior, sequence_posterior, coherence_score
4658:     )
4659:
4660:     # Detect gaps
4661:     gaps = self._detect_gaps(node, observations, uncertainty)
4662:
4663:     return {
4664:         'mechanism_type': mechanism_type_posterior,
4665:         'activity_sequence': sequence_posterior,
4666:         'coherence_score': coherence_score,
4667:         'sufficiency_test': sufficiency,
4668:         'necessity_test': necessity,
4669:         'uncertainty': uncertainty,
4670:         'gaps': gaps,
4671:         'observations': observations
4672:     }
4673:
4674: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._extract_observations")
4675: def _extract_observations(self, node: MetaNode, text: str) -> dict[str, Any]:
4676:     """Extract textual observations related to the mechanism"""
4677:     # Find node context in text
4678:     node_pattern = re.escape(node.id)
4679:     matches = list(re.finditer(node_pattern, text, re.IGNORECASE))
4680:
4681:     observations = {
4682:         'entity_activity': None,
```

```

4683:         'verbs': [],
4684:         'entities': [],
4685:         'budget': node.financial_allocation,
4686:         'context_snippets': []
4687:     }
4688:
4689:     if node.entity_activity:
4690:         observations['entity_activity'] = {
4691:             'entity': node.entity_activity.entity,
4692:             'activity': node.entity_activity.activity,
4693:             'verb_lemma': node.entity_activity.verb_lemma
4694:         }
4695:
4696:     # Extract context around node mentions
4697:     for match in matches[:3]: # Limit to first 3 occurrences
4698:         start = max(0, match.start() - 300)
4699:         end = min(len(text), match.end() + 300)
4700:         context = text[start:end]
4701:
4702:         # Process with spaCy
4703:         doc = self.nlp(context)
4704:
4705:         # Extract verbs
4706:         verbs = [token.lemma_ for token in doc if token.pos_ == 'VERB']
4707:         observations['verbs'].extend(verbs)
4708:
4709:         # Extract entities
4710:         entities = [ent.text for ent in doc.ents if ent.label_ in ['ORG', 'PER']]
4711:         observations['entities'].extend(entities)
4712:
4713:         observations['context_snippets'].append(context[:200])
4714:
4715:     return observations
4716:
4717: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_mechanism_type")
4718: def _infer_mechanism_type(self, observations: dict[str, Any]) -> dict[str, float]:
4719:     """Infer mechanism type using Bayesian updating"""
4720:     # Start with hyperprior
4721:     posterior = dict(self.mechanism_type_priors)
4722:
4723:     # Get Laplace smoothing parameter from configuration
4724:     laplace_smooth = self.config.get_bayesian_threshold('laplace_smoothing')
4725:
4726:     # Update based on observed verbs
4727:     observed_verbs = set(observations.get('verbs', []))
4728:
4729:     if observed_verbs:
4730:         for mech_type, typical_verbs in self.mechanism_sequences.items():
4731:             # Count overlap
4732:             overlap = len(observed_verbs.intersection(set(typical_verbs)))
4733:             total = len(typical_verbs)
4734:
4735:             if total > 0:
4736:                 # Likelihood: proportion of typical verbs observed with Laplace smoothing
4737:                 likelihood = (overlap + laplace_smooth) / (total + 2 * laplace_smooth)
4738:

```

```

4739:             # Bayesian update
4740:             posterior[mech_type] *= likelihood
4741:
4742:             # Update based on entity-activity
4743:             if observations.get('entity_activity'):
4744:                 verb = observations['entity_activity'].get('verb_lemma', '')
4745:                 for mech_type, typical_verbs in self.mechanism_sequences.items():
4746:                     if verb in typical_verbs:
4747:                         posterior[mech_type] *= 1.5
4748:
4749:             # Normalize
4750:             total = sum(posterior.values())
4751:             if total > 0:
4752:                 posterior = {k: v / total for k, v in posterior.items()}
4753:
4754:             return posterior
4755:
4756:     @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._infer_activity_sequence")
4757:     def _infer_activity_sequence(self, observations: dict[str, Any],
4758:                                 mechanism_type_posterior: dict[str, float]) -> dict[str, Any]:
4759:         """Infer activity sequence parameters"""
4760:         # Get most likely mechanism type
4761:         best_type = max(mechanism_type_posterior.items(), key=lambda x: x[1])[0]
4762:         expected_sequence = self.mechanism_sequences.get(best_type, [])
4763:
4764:         observed_verbs = observations.get('verbs', [])
4765:
4766:         # Calculate transition probabilities (simplified Markov chain)
4767:         transitions = {}
4768:         for i in range(len(expected_sequence) - 1):
4769:             current = expected_sequence[i]
4770:             next_verb = expected_sequence[i + 1]
4771:
4772:             # Check if transition is observed
4773:             if current in observed_verbs and next_verb in observed_verbs:
4774:                 transitions[(current, next_verb)] = 0.85
4775:             else:
4776:                 transitions[(current, next_verb)] = 0.40
4777:
4778:         return {
4779:             'expected_sequence': expected_sequence,
4780:             'observed_verbs': observed_verbs,
4781:             'transition_probabilities': transitions,
4782:             'sequence_completeness': len(set(observed_verbs) & set(expected_sequence)) / max(len(expected_sequence), 1)
4783:         }
4784:
4785:     @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_coherence_factor")
4786:     def _calculate_coherence_factor(self, node: MetaNode,
4787:                                     observations: dict[str, Any],
4788:                                     all_nodes: dict[str, MetaNode]) -> float:
4789:         """Calculate mechanism coherence score"""
4790:         coherence = 0.0 # Refactored
4791:         weights = []
4792:
4793:         # Factor 1: Entity-Activity presence
4794:         if observations.get('entity_activity'):

```

```
4795:         coherence += 0.30
4796:         weights.append(0.30)
4797:
4798:     # Factor 2: Budget consistency
4799:     if observations.get('budget'):
4800:         coherence += 0.20
4801:         weights.append(0.20)
4802:
4803:     # Factor 3: Verb sequence completeness
4804:     seq_info = observations.get('verbs', [])
4805:     if seq_info:
4806:         verb_score = min(len(seq_info) / 4.0, 1.0)  # Expect ~4 verbs
4807:         coherence += verb_score * 0.25
4808:         weights.append(0.25)
4809:
4810:     # Factor 4: Entity presence
4811:     if observations.get('entities'):
4812:         coherence += 0.15
4813:         weights.append(0.15)
4814:
4815:     # Factor 5: Context richness
4816:     snippets = observations.get('context_snippets', [])
4817:     if snippets:
4818:         coherence += 0.10
4819:         weights.append(0.10)
4820:
4821:     # Normalize by actual weights used
4822:     if weights:
4823:         coherence = coherence / sum(weights) if sum(weights) > 0 else 0.0
4824:
4825:     return coherence
4826:
4827: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._test_sufficiency")
4828: def _test_sufficiency(self, node: MetaNode,
4829:                       observations: dict[str, Any]) -> dict[str, Any]:
4830:     """Test if mechanism is sufficient to produce the outcome"""
4831:     # Check if entity has capability
4832:     has_entity = observations.get('entity_activity') is not None
4833:
4834:     # Check if activities are present
4835:     has_activities = len(observations.get('verbs', [])) >= 2
4836:
4837:     # Check if resources are allocated
4838:     has_resources = observations.get('budget') is not None
4839:
4840:     sufficiency_score = (
4841:         (0.4 if has_entity else 0.0) +
4842:         (0.4 if has_activities else 0.0) +
4843:         (0.2 if has_resources else 0.0)
4844:     )
4845:
4846:     return {
4847:         'score': sufficiency_score,
4848:         'is_sufficient': sufficiency_score >= 0.6,
4849:         'components': {
4850:             'entity': has_entity,
```

```
4851:             'activities': has_activities,
4852:             'resources': has_resources
4853:         }
4854:     }
4855:
4856:     @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._test_necessity")
4857:     def _test_necessity(self, node: MetaNode,
4858:                         observations: dict[str, Any]) -> dict[str, Any]:
4859:         """
4860:             AUDIT POINT 2.2: Mechanism Necessity Hoop Test
4861:
4862:             Test if mechanism is necessary by checking documented components:
4863:             - Entity (responsable)
4864:             - Activity (verb lemma sequence)
4865:             - Budget (presupuesto asignado)
4866:
4867:             Implements Beach 2017 Hoop Tests for necessity verification.
4868:             Per Falletti & Lynch 2009, Bayesian-deterministic hybrid boosts mechanism depth.
4869:
4870:             Returns:
4871:                 Dict with 'is_necessary', 'missing_components', and remediation text
4872:             """
4873:             # F1.2: Use refactored NecessitySufficiencyTester if available
4874:             if self.bayesian_adapter and self.bayesian_adapter.necessity_tester:
4875:                 try:
4876:                     return self.bayesian_adapter.test_necessity_from_observations(
4877:                         node.id,
4878:                         observations
4879:                     )
4880:                 except Exception as e:
4881:                     self.logger.warning(f"Error en tester refactorizado: {e}, usando legacy")
4882:
4883:             # AUDIT POINT 2.2: Enhanced necessity test with documented components
4884:             missing_components = []
4885:
4886:             # 1. Check Entity documentation
4887:             entities = observations.get('entities', [])
4888:             entity_activity = observations.get('entity_activity')
4889:
4890:             if not entity_activity or not entity_activity.get('entity'):
4891:                 missing_components.append('entity')
4892:             else:
4893:                 # Verify unique entity (not multiple conflicting entities)
4894:                 unique_entity = len(set(entities)) == 1 if entities else False
4895:                 if not unique_entity and len(entities) > 1:
4896:                     missing_components.append('unique_entity')
4897:
4898:             # 2. Check Activity documentation (verb lemma sequence)
4899:             verbs = observations.get('verbs', [])
4900:             if not verbs or len(verbs) < 1:
4901:                 missing_components.append('activity')
4902:             else:
4903:                 # Check for specific action verbs (not just generic ones)
4904:                 specific_verbs = [v for v in verbs if v in [
4905:                     'implementar', 'ejecutar', 'realizar', 'desarrollar',
4906:                     'construir', 'diseñar', 'planificar', 'coordinar',
```

```

4907:             'gestionar', 'supervisar', 'controlar', 'auditar'
4908:         ]
4909:     if not specific_verbs:
4910:         missing_components.append('specific_activity')
4911:
4912:     # 3. Check Budget documentation
4913:     budget = observations.get('budget')
4914:     if budget is None or budget <= 0:
4915:         missing_components.append('budget')
4916:
4917:     # Calculate necessity score
4918:     # All three components must be present for necessity=True
4919:     is_necessary = len(missing_components) == 0
4920:
4921:     # Calculate partial score for reporting
4922:     max_components = 3 # entity, activity, budget
4923:     present_components = max_components - len(
4924:         [c for c in missing_components if c in ['entity', 'activity', 'budget']])
4925:     necessity_score = present_components / max_components
4926:
4927:     result = {
4928:         'score': necessity_score,
4929:         'is_necessary': is_necessary,
4930:         'missing_components': missing_components,
4931:         'alternatives_likely': not is_necessary,
4932:         'hoop_test_passed': is_necessary
4933:     }
4934:
4935:     # Add remediation text if test fails
4936:     if not is_necessary:
4937:         result['remediation'] = self._generate_necessity_remediation(node.id, missing_components)
4938:
4939:     return result
4940:
4941: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._generate_necessity_remediation")
4942: def _generate_necessity_remediation(self, node_id: str, missing_components: list[str]) -> str:
4943:     """Generate remediation text for failed necessity test"""
4944:     component_descriptions = {
4945:         'entity': 'entidad responsable claramente identificada',
4946:         'unique_entity': 'una \'unica entidad responsable (m\'ultiples entidades detectadas)',
4947:         'activity': 'secuencia de actividades documentada',
4948:         'specific_activity': 'actividades espec\'ificas (no gen\'ericas)',
4949:         'budget': 'presupuesto asignado y cuantificado'
4950:     }
4951:
4952:     missing_desc = ', '.join([component_descriptions.get(c, c) for c in missing_components])
4953:
4954:     return (
4955:         f"\"Mecanismo para {node_id} falla Hoop Test de necesidad (D6-Q2). \""
4956:         f"\"Componentes faltantes: {missing_desc}. \""
4957:         f"\"Se requiere documentar estos componentes necesarios para validar \""
4958:         f"\"la cadena causal seg\'un Beach 2017.\""
4959:     )
4960:
4961: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty")
4962: def _quantify_uncertainty(self, mechanism_type_posterior: dict[str, float],

```

```

4963:             sequence_posterior: dict[str, Any],
4964:             coherence_score: float) -> dict[str, float]:
4965:             """Quantify epistemic uncertainty"""
4966:             # Entropy of mechanism type distribution
4967:             mech_probs = list(mechanism_type_posterior.values())
4968:             if mech_probs:
4969:                 mech_entropy = -sum(p * np.log(p + 1e-10) for p in mech_probs if p > 0)
4970:                 max_entropy = np.log(len(mech_probs))
4971:                 mech_uncertainty = mech_entropy / max_entropy if max_entropy > 0 else ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_mech_uncertainty", 1.0)
4972:             else:
4973:                 mech_uncertainty = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_mech_uncertainty", 1.0) # Refactored
4974:
4975:             # Sequence completeness uncertainty
4976:             seq_completeness = sequence_posterior.get('sequence_completeness', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "default_seq_completeness", 0.0))
4977:             seq_uncertainty = 1.0 - seq_completeness
4978:
4979:             # Coherence uncertainty
4980:             coherence_uncertainty = 1.0 - coherence_score
4981:
4982:             # Combined uncertainty
4983:             total_uncertainty = (
4984:                 mech_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "mech_uncertainty_weight", 0.4) +
4985:                 seq_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "seq_uncertainty_weight", 0.3) +
4986:                 coherence_uncertainty * ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty", "coherence_uncertainty_weight", 0.3)
4987:             )
4988:
4989:             return {
4990:                 'total': total_uncertainty,
4991:                 'mechanism_type': mech_uncertainty,
4992:                 'sequence': seq_uncertainty,
4993:                 'coherence': coherence_uncertainty
4994:             }
4995:
4996:             @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._detect_gaps")
4997:             def _detect_gaps(self, node: MetaNode, observations: dict[str, Any],
4998:                             uncertainty: dict[str, float]) -> list[dict[str, str]]:
4999:                 """Detect documentation gaps based on uncertainty"""
5000:                 gaps = []
5001:
5002:                 # High total uncertainty
5003:                 if uncertainty['total'] > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._detect_gaps", "high_uncertainty_threshold", 0.6):
5004:                     gaps.append({
5005:                         'type': 'high_uncertainty',
5006:                         'severity': 'high',
5007:                         'message': f'Mecanismo para {node.id} tiene alta incertidumbre ({uncertainty["total"]:.2f})',
5008:                         'suggestion': "Se requiere más documentación sobre el mecanismo causal"
5009:                     })
5010:
5011:                 # Missing entity

```

```

5012:         if not observations.get('entity_activity'):
5013:             gaps.append({
5014:                 'type': 'missing_entity',
5015:                 'severity': 'high',
5016:                 'message': f"No se especifica entidad responsable para {node.id}",
5017:                 'suggestion': "Especificar quÃ© entidad ejecutarÃ¡ las actividades"
5018:             })
5019:
5020:     # Insufficient activities
5021:     if len(observations.get('verbs', [])) < 2:
5022:         gaps.append({
5023:             'type': 'insufficient_activities',
5024:             'severity': 'medium',
5025:             'message': f"Few activities documented for {node.id}",
5026:             'suggestion': "Detail the necessary activities to achieve the product"
5027:         })
5028:
5029:     # Missing budget
5030:     if not observations.get('budget'):
5031:         gaps.append({
5032:             'type': 'missing_budget',
5033:             'severity': 'medium',
5034:             'message': f"Missing budgetary assignment for {node.id}",
5035:             'suggestion': "Assign financial resources to the product"
5036:         })
5037:
5038:     return gaps
5039:
5040: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._aggregate_bayesian_confidence")
5041: def _aggregate_bayesian_confidence(self, confidences: list[float]) -> float:
5042:     """
5043:         Aggregate multiple Bayesian confidence values.
5044:
5045:     Args:
5046:         confidences: List of confidence values to aggregate
5047:
5048:     Returns:
5049:         Aggregated confidence value
5050:     """
5051:     if not confidences:
5052:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._aggregate_bayesian_confidence", "default_confidence",
0.5) # Default neutral confidence
5053:     return float(np.mean(confidences))
5054:
5055: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix")
5056: def _build_transition_matrix(self, mechanism_type: str) -> np.ndarray:
5057:     """
5058:         Build transition matrix for activity sequences.
5059:
5060:     Args:
5061:         mechanism_type: Type of mechanism
5062:
5063:     Returns:
5064:         Transition probability matrix
5065:     """
5066:     # Get typical sequence for this mechanism type

```

```

5067:     sequence = self.mechanism_sequences.get(mechanism_type, ['planificar', 'ejecutar', 'evaluar'])
5068:     n = len(sequence)
5069:
5070:     # Create a simple sequential transition matrix
5071:     matrix = np.zeros((n, n))
5072:     for i in range(n - 1):
5073:         matrix[i, i + 1] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "next_step_prob", 0.7) # High probability of next step
5074:         matrix[i, i] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "stay_prob", 0.2)
5075:     # Some probability of staying in same step
5076:     if i < n - 2:
5077:         matrix[i, i + 2] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "skip_prob", 0.1) # Small probability of skipping
5078:     matrix[n - 1, n - 1] = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._build_transition_matrix", "absorbing_prob", 1.0) # Final state is absorbing
5079:     return matrix
5080:
5081: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior")
5082: def _calculate_type_transition_prior(self, from_type: str, to_type: str) -> float:
5083:     """
5084:     Calculate prior probability of transitioning between mechanism types.
5085:
5086:     Args:
5087:         from_type: Source mechanism type
5088:         to_type: Target mechanism type
5089:
5090:     Returns:
5091:         Prior probability of transition
5092:     """
5093:     # Same type has high probability
5094:     if from_type == to_type:
5095:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "same_type_prob", 0.7)
5096:
5097:     # Related types have medium probability
5098:     related_pairs = [
5099:         ('administrativo', 'politico'),
5100:         ('tecnico', 'financiero'),
5101:         ('financiero', 'administrativo'),
5102:     ]
5103:     if (from_type, to_type) in related_pairs or (to_type, from_type) in related_pairs:
5104:         return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "related_type_prob", 0.2)
5105:
5106:     # Unrelated types have low probability
5107:     return ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._calculate_type_transition_prior", "unrelated_type_prob", 0.1)
5108:
5109: @calibrated_method("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type")
5110: def _classify_mechanism_type(self, observations: dict[str, Any]) -> str:
5111:     """
5112:     Classify mechanism type based on observations.
5113:
5114:     Args:
5115:         observations: Observed features

```

```

5116:
5117:     Returns:
5118:         Classified mechanism type
5119:     """
5120:     # Extract features
5121:     verbs = observations.get('verbs', [])
5122:     entities = observations.get('entities', [])
5123:     budget = observations.get('budget')
5124:
5125:     # Score each mechanism type
5126:     scores = {}
5127:     for mech_type, typical_verbs in self.mechanism_sequences.items():
5128:         score = ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "score", 0.0) # Refactored
5129:         # Count matching verbs
5130:         for verb in verbs:
5131:             if any(tv in verb.lower() for tv in typical_verbs):
5132:                 score += ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type", "verb_match_bonus"
", 1.0)
5133:             scores[mech_type] = score
5134:
5135:     # Adjust for budget presence (indicates financial mechanism)
5136:     if budget and budget > 0:
5137:         scores['financiero'] = scores.get('financiero', 0) + 2.0
5138:
5139:     # Adjust for political/administrative entities
5140:     for entity in entities:
5141:         entity_lower = entity.lower()
5142:         if any(word in entity_lower for word in ['alcaldÃ-a', 'consejo', 'gobernaciÃ³n']):
5143:             scores['politico'] = scores.get('politico', 0) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInference._classif
y_mechanism_type", "political_entity_bonus", 1.0)
5144:             if any(word in entity_lower for word in ['secretarÃ-a', 'direcciÃ³n', 'oficina']):
5145:                 scores['administrativo'] = scores.get('administrativo', 0) + ParameterLoaderV2.get("farfan_core.analysis.derek_beach.BayesianMechanismInfere
nce._classify_mechanism_type", "administrative_entity_bonus", 1.0)
5146:
5147:     # Return type with highest score, or 'mixto' if tie
5148:     if not scores or all(s == 0 for s in scores.values()):
5149:         return 'mixto'
5150:
5151:     max_score = max(scores.values())
5152:     max_types = [t for t, s in scores.items() if s == max_score]
5153:
5154:     if len(max_types) > 1:
5155:         return 'mixto'
5156:     return max_types[0]
5157:
5158: class CausalInferenceSetup:
5159:     """Prepare model for causal inference"""
5160:
5161:     def __init__(self, config: ConfigLoader) -> None:
5162:         self.logger = logging.getLogger(self.__class__.__name__)
5163:         self.config = config
5164:         self.goal_classification = config.get('lexicons.goal_classification', {})
5165:         self.admin_keywords = config.get('lexicons.administrative_keywords', [])
5166:         self.contextual_factors = config.get('lexicons.contextual_factors', [])
5167:
5168:     @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.classify_goal_dynamics")

```

```

5169:     def classify_goal_dynamics(self, nodes: dict[str, MetaNode]) -> None:
5170:         """Classify dynamics for each goal"""
5171:         for node in nodes.values():
5172:             text_lower = node.text.lower()
5173:
5174:             for keyword, dynamics in self.goal_classification.items():
5175:                 if keyword in text_lower:
5176:                     node.dynamics = cast("DynamicsType", dynamics)
5177:                     self.logger.debug(f"Meta {node.id} clasificada como {node.dynamics}")
5178:                     break
5179:
5180:     @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.assign_probative_value")
5181:     def assign_probative_value(self, nodes: dict[str, MetaNode]) -> None:
5182:         """Assign probative test types to nodes"""
5183:         # Import INDICATOR_STRUCTURE from financiero_viability_tablas
5184:         try:
5185:             from financiero_viability_tablas import ColombianMunicipalContext
5186:             indicator_structure = ColombianMunicipalContext.INDICATOR_STRUCTURE
5187:         except ImportError:
5188:             indicator_structure = {
5189:                 'resultado': ['lÃ±ea_base', 'meta', 'aÃ±o_base', 'aÃ±o_meta', 'fuente', 'responsable'],
5190:                 'producto': ['indicador', 'fÃ³rmula', 'unidad_medida', 'lÃ±ea_base', 'meta', 'periodicidad'],
5191:                 'gestiÃ³n': ['eficacia', 'eficiencia', 'economÃ-a', 'costo_beneficio']
5192:             }
5193:
5194:         for node in nodes.values():
5195:             text_lower = node.text.lower()
5196:
5197:             # Cross-reference with INDICATOR_STRUCTURE to classify critical requirements
5198:             # as Hoop Tests or Smoking Guns
5199:             indicator_structure.get(node.type, [])
5200:
5201:             # Check if node has all critical DNP requirements (D3-Q1 indicators)
5202:             has_linea_base = bool(
5203:                 node.baseline and str(node.baseline).upper() not in ['ND', 'POR DEFINIR', 'N/A', 'NONE'])
5204:             has_meta = bool(node.target and str(node.target).upper() not in ['ND', 'POR DEFINIR', 'N/A', 'NONE'])
5205:             has_fuente = 'fuente' in text_lower or 'fuente de informaciÃ³n' in text_lower
5206:
5207:             # Perfect Hoop Test: Missing any critical requirement = total hypothesis failure
5208:             # This applies to producto nodes with D3-Q1 indicators
5209:             if node.type == 'producto':
5210:                 if has_linea_base and has_meta and has_fuente:
5211:                     # Perfect indicators trigger Hoop Test classification
5212:                     node.test_type = 'hoop_test'
5213:                     self.logger.debug(f"Meta {node.id} classified as hoop_test (perfect D3-Q1 compliance)")
5214:                 elif not has_linea_base or not has_meta:
5215:                     # Missing critical requirements - still Hoop Test but will fail
5216:                     node.test_type = 'hoop_test'
5217:                     node.audit_flags.append('hoop_test_failure')
5218:                     self.logger.warning(f"Meta {node.id} FAILS hoop_test (missing D3-Q1 critical fields)")
5219:                 else:
5220:                     node.test_type = 'straw_in_wind'
5221:             # Check for administrative/regulatory nature (Hoop Test)
5222:             elif any(keyword in text_lower for keyword in self.admin_keywords):
5223:                 node.test_type = 'hoop_test'
5224:             # Check for highly specific outcomes (Smoking Gun)

```

```

5225:         elif node.type == 'resultado' and node.target and node.baseline:
5226:             try:
5227:                 float(str(node.target).replace(',', '').replace('%', ''))
5228:                 # Smoking Gun: rare, highly specific evidence with strong inferential power
5229:                 node.test_type = 'smoking_gun'
5230:             except (ValueError, TypeError):
5231:                 node.test_type = 'straw_in_wind'
5232:             # Double decisive for critical impact goals
5233:             elif node.type == 'impacto' and node.rigor_status == 'fuerte':
5234:                 node.test_type = 'doubly_decisional'
5235:             else:
5236:                 node.test_type = 'straw_in_wind'
5237:
5238:             self.logger.debug(f"Meta {node.id} assigned test type: {node.test_type}")
5239:
5240:     @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup.identify_failure_points")
5241:     def identify_failure_points(self, graph: nx.DiGraph, text: str) -> set[str]:
5242:         """Identify single points of failure in causal chain
5243:
5244:             Harmonic Front 3 - Enhancement 2: Contextual Failure Point Detection
5245:             Expands risk_pattern to explicitly include localized contextual factors from rubrics:
5246:             - restricciones territoriales
5247:             - patrones culturales machistas
5248:             - limitaciÃ³n normativa
5249:
5250:             For D6-Q5 (Enfoque Diferencial/Restricciones): Excelente requires a minimum of 3 distinct
5251:             contextual factors correctly mapped to nodes, satisfying enfoque_diferencial
5252:             and analisis_contextual criteria.
5253:             """
5254:             failure_points = set()
5255:
5256:             # Find nodes with high out-degree (many dependencies)
5257:             for node_id in graph.nodes():
5258:                 out_degree = graph.out_degree(node_id)
5259:                 node_type = graph.nodes[node_id].get('type')
5260:
5261:                 if node_type == 'producto' and out_degree >= 3:
5262:                     failure_points.add(node_id)
5263:                     self.logger.warning(f"Punto Ã³nico de falla identificado: {node_id} "
5264:                                         f"(grado de salida: {out_degree})")
5265:
5266:             # HARMONIC FRONT 3 - Enhancement 2: Expand contextual factors
5267:             # Add specific rubric factors for D6-Q5 compliance
5268:             extended_contextual_factors = list(self.contextual_factors) + [
5269:                 'restricciones territoriales',
5270:                 'restricciÃ³n territorial',
5271:                 'limitaciÃ³n territorial',
5272:                 'patrones culturales machistas',
5273:                 'machismo',
5274:                 'inequidad de gÃ©nero',
5275:                 'violencia de gÃ©nero',
5276:                 'limitaciÃ³n normativa',
5277:                 'limitaciÃ³n legal',
5278:                 'restricciÃ³n legal',
5279:                 'barrera institucional',
5280:                 'restricciÃ³n presupuestal',

```

```
5281:         'ausencia de capacidad tÃ©cnica',
5282:         'baja capacidad institucional',
5283:         'conflicto armado',
5284:         'desplazamiento forzado',
5285:         'poblaciÃ³n dispersa',
5286:         'ruralidad dispersa',
5287:         'acceso vial limitado',
5288:         'conectividad deficiente'
5289:     ]
5290:
5291:     # Extract contextual risks from text
5292:     risk_pattern = '|'.join(re.escape(factor) for factor in extended_contextual_factors)
5293:     risk_regex = re.compile(rf'\b({risk_pattern})\b', re.IGNORECASE)
5294:
5295:     # Track distinct contextual factors for D6-Q5 quality criteria
5296:     contextual_factors_detected = set()
5297:     node_contextual_map = defaultdict(set)
5298:
5299:     # Find risk mentions and associate with nodes
5300:     for match in risk_regex.finditer(text):
5301:         risk_text = match.group()
5302:         contextual_factors_detected.add(risk_text.lower())
5303:
5304:         context_start = max(0, match.start() - 200)
5305:         context_end = min(len(text), match.end() + 200)
5306:         context = text[context_start:context_end]
5307:
5308:         # Try to find node mentions in risk context
5309:         for node_id in graph.nodes():
5310:             if node_id in context:
5311:                 failure_points.add(node_id)
5312:                 if 'contextual_risks' not in graph.nodes[node_id]:
5313:                     graph.nodes[node_id]['contextual_risks'] = []
5314:                     graph.nodes[node_id]['contextual_risks'].append(risk_text)
5315:                     node_contextual_map[node_id].add(risk_text.lower())
5316:
5317:     # D6-Q5 quality criteria assessment
5318:     distinct_factors_count = len(contextual_factors_detected)
5319:     d6_q5_quality = 'insuficiente'
5320:     if distinct_factors_count >= 3:
5321:         d6_q5_quality = 'excelente'
5322:     elif distinct_factors_count >= 2:
5323:         d6_q5_quality = 'bueno'
5324:     elif distinct_factors_count >= 1:
5325:         d6_q5_quality = 'aceptable'
5326:
5327:     # Store D6-Q5 metrics in graph attributes
5328:     graph.graph['d6_q5_contextual_factors'] = list(contextual_factors_detected)
5329:     graph.graph['d6_q5_distinct_count'] = distinct_factors_count
5330:     graph.graph['d6_q5_quality'] = d6_q5_quality
5331:     graph.graph['d6_q5_node_mapping'] = dict(node_contextual_map)
5332:
5333:     self.logger.info(f"Puntos de falla identificados: {len(failure_points)}")
5334:     self.logger.info(
5335:         f"D6-Q5: {distinct_factors_count} factores contextuales distintos detectados - {d6_q5_quality}")
5336:
```

```
5337:         return failure_points
5338:
5339:     @calibrated_method("farfan_core.analysis.derek_beach.CausalInferenceSetup._get_dynamics_pattern")
5340:     def _get_dynamics_pattern(self, dynamics_type: str) -> str:
5341:         """
5342:             Get the pattern associated with a dynamics type.
5343:
5344:             Args:
5345:                 dynamics_type: Type of dynamics (suma, decreciente, constante, indefinido)
5346:
5347:             Returns:
5348:                 Pattern string for the dynamics type
5349:             """
5350:             patterns = {
5351:                 'suma': 'suma|total|agregado|consolidado',
5352:                 'decreciente': 'reducir|disminuir|decrementar|bajar',
5353:                 'constante': 'mantener|sostener|preservar|conservar',
5354:                 'indefinido': 'por definir|sin especificar|indefinido'
5355:             }
5356:             return patterns.get(dynamics_type, '')
5357:
5358: class ReportingEngine:
5359:     """Generate visualizations and reports"""
5360:
5361:     def __init__(self, config: ConfigLoader, output_dir: Path) -> None:
5362:         self.logger = logging.getLogger(self.__class__.__name__)
5363:         self.config = config
5364:         self.output_dir = output_dir
5365:         self.output_dir.mkdir(parents=True, exist_ok=True)
5366:
5367:     @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram")
5368:     def generate_causal_diagram(self, graph: nx.DiGraph, policy_code: str) -> Path:
5369:         """Generate causal diagram visualization"""
5370:         dot = Dot(graph_type='digraph', rankdir='TB')
5371:         dot.set_name(f'{policy_code}_causal_model')
5372:         dot.set_node_defaults(
5373:             shape='box',
5374:             style='rounded,filled',
5375:             fontname='Arial',
5376:             fontsize='10'
5377:         )
5378:         dot.set_edge_defaults(
5379:             fontsize='8',
5380:             fontname='Arial'
5381:         )
5382:
5383:         # Add nodes with rigor coloring
5384:         for node_id in graph.nodes():
5385:             node_data = graph.nodes[node_id]
5386:
5387:             # Determine color based on rigor status and audit flags
5388:             rigor = node_data.get('rigor_status', 'sin_evaluar')
5389:             audit_flags = node_data.get('audit_flags', [])
5390:             financial = node_data.get('financial_allocation')
5391:
5392:             if rigor == 'dÃ©bil' or not financial:
```

```

5393:         color = 'lightcoral' # Red
5394:     elif audit_flags:
5395:         color = 'lightyellow' # Yellow
5396:     else:
5397:         color = 'lightgreen' # Green
5398:
5399:     # Create label
5400:     node_type = node_data.get('type', 'programa')
5401:     text = node_data.get('text', '')[:80]
5402:     label = f"{node_id}\n{n[{node_type.upper()}]\n{text}...}"
5403:
5404:     entity = node_data.get('responsible_entity')
5405:     if entity:
5406:         label += f"\nδ\237\221 {entity[:30]}"
5407:
5408:     if financial:
5409:         label += f"\nδ\237\222° ${financial:.0f}"
5410:
5411:     dot_node = Node(
5412:         node_id,
5413:         label=label,
5414:         fillcolor=color
5415:     )
5416:     dot.add_node(dot_node)
5417:
5418:     # Add edges with causal logic
5419:     for source, target in graph.edges():
5420:         edge_data = graph.edges[source, target]
5421:         keyword = edge_data.get('keyword', '')
5422:         strength = edge_data.get('strength', ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram", "default_strength", 0.5))
5423:
5424:         # Determine edge style based on strength
5425:         style = 'solid' if strength > ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_diagram", "solid_strength_threshold", 0.7) else 'dashed'
5426:
5427:         dot_edge = Edge(
5428:             source,
5429:             target,
5430:             label=keyword[:20],
5431:             style=style
5432:         )
5433:         dot.add_edge(dot_edge)
5434:
5435:     # Save files
5436:     dot_path = self.output_dir / f"{policy_code}_causal_diagram.dot"
5437:     png_path = self.output_dir / f"{policy_code}_causal_diagram.png"
5438:
5439:     try:
5440:         with open(dot_path, "w", encoding="utf-8") as f:
5441:             f.write(dot.to_string())
5442:             self.logger.info(f"Diagrama DOT guardado en: {dot_path}")
5443:
5444:         # Try to render PNG
5445:         try:
5446:             dot.write_png(str(png_path))

```

```

5447:         self.logger.info(f"Diagrama PNG renderizado en: {png_path}")
5448:     except Exception as e:
5449:         self.logger.warning(f"No se pudo renderizar PNG (¿Graphviz instalado?): {e}")
5450:     except Exception as e:
5451:         self.logger.error(f"Error guardando diagrama: {e}")
5452:
5453:     return png_path
5454:
5455: @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_accountability_matrix")
5456: def generate_accountability_matrix(self, graph: nx.DiGraph,
5457:                                     policy_code: str) -> Path:
5458:     """Generate accountability matrix in Markdown"""
5459:     md_path = self.output_dir / f"{policy_code}_accountability_matrix.md"
5460:
5461:     # Group by impact goals
5462:     impact_goals = [n for n in graph.nodes()
5463:                      if graph.nodes[n].get('type') == 'impacto']
5464:
5465:     content = [f"# Matriz de Responsabilidades - {policy_code}\n"]
5466:     content.append("*Generado automáticamente por CDAF v2.0*\n")
5467:     content.append("---\n")
5468:
5469:     for impact in impact_goals:
5470:         impact_data = graph.nodes[impact]
5471:         content.append(f"\# Meta de Impacto: {impact}\n")
5472:         content.append(f"**Descripción:** {impact_data.get('text', 'N/A')}\n\n")
5473:
5474:         # Find all predecessor chains
5475:         predecessors = list(nx.ancestors(graph, impact))
5476:
5477:         if predecessors:
5478:             content.append(" | Meta | Tipo | Entidad Responsable | Actividad Clave | Presupuesto |\n")
5479:             content.append(" |-----| -----|-----|-----|-----|\n")
5480:
5481:             for pred in predecessors:
5482:                 pred_data = graph.nodes[pred]
5483:                 meta_type = pred_data.get('type', 'N/A')
5484:                 entity = pred_data.get('responsible_entity', 'No asignado')
5485:
5486:                 ea = pred_data.get('entity_activity')
5487:                 activity = 'N/A'
5488:                 if ea and isinstance(ea, dict):
5489:                     activity = ea.get('activity', 'N/A')
5490:
5491:                 budget = pred_data.get('financial_allocation')
5492:                 budget_str = f"${{budget:.0f}}" if budget else "Sin presupuesto"
5493:
5494:                 content.append(f" | {pred} | {meta_type} | {entity} | {activity} | {budget_str} | \n")
5495:
5496:             content.append("\n")
5497:         else:
5498:             content.append("*No se encontraron metas intermedias.*\n\n")
5499:
5500:         content.append("\n---\n")
5501:         content.append("### Leyenda\n")
5502:         content.append("- **Meta de Impacto:** Resultado final esperado\n")

```

```

5503:     content.append("- **Meta de Resultado:** Cambio intermedio observable\n")
5504:     content.append("- **Meta de Producto:** Entrega tangible del programa\n")
5505:
5506:     try:
5507:         with open(md_path, 'w', encoding='utf-8') as f:
5508:             f.write(''.join(content))
5509:             self.logger.info(f"Matriz de responsabilidades guardada en: {md_path}")
5510:     except Exception as e:
5511:         self.logger.error(f"Error guardando matriz de responsabilidades: {e}")
5512:
5513:     return md_path
5514:
5515: @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_confidence_report")
5516: def generate_confidence_report(self,
5517:                                 nodes: dict[str, MetaNode],
5518:                                 graph: nx.DiGraph,
5519:                                 causal_chains: list[CausalLink],
5520:                                 audit_results: dict[str, AuditResult],
5521:                                 financial_auditor: FinancialAuditor,
5522:                                 sequence_warnings: list[str],
5523:                                 policy_code: str) -> Path:
5524:     """Generate extraction confidence report"""
5525:     json_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"
5526:
5527:     # Calculate metrics
5528:     total_metas = len(nodes)
5529:
5530:     metas_with_ea = sum(1 for n in nodes.values() if n.entity_activity)
5531:     metas_with_ea_pct = (metas_with_ea / total_metas * 100) if total_metas > 0 else 0
5532:
5533:     enlaces_with_logic = sum(1 for link in causal_chains if link.get('logic'))
5534:     total_edges = graph.number_of_edges()
5535:     enlaces_with_logic_pct = (enlaces_with_logic / total_edges * 100) if total_edges > 0 else 0
5536:
5537:     metas_passed_audit = sum(1 for r in audit_results.values() if r['passed'])
5538:     metas_with_traceability_pct = (metas_passed_audit / total_metas * 100) if total_metas > 0 else 0
5539:
5540:     metas_with_financial = sum(1 for n in nodes.values() if n.financial_allocation)
5541:     metas_with_financial_pct = (metas_with_financial / total_metas * 100) if total_metas > 0 else 0
5542:
5543:     # Node type distribution
5544:     type_distribution = defaultdict(int)
5545:     for node in nodes.values():
5546:         type_distribution[node.type] += 1
5547:
5548:     # Rigor distribution
5549:     rigor_distribution = defaultdict(int)
5550:     for node in nodes.values():
5551:         rigor_distribution[node.rigor_status] += 1
5552:
5553:     report = {
5554:         "metadata": {
5555:             "policy_code": policy_code,
5556:             "framework_version": "2." + str(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_confidence_report", "framework_version", 0)),
5557:             "total_nodes": total_metas,

```

```

5558:         "total_edges": total_edges
5559:     },
5560:     "extraction_metrics": {
5561:         "total_metas_identificadas": total_metas,
5562:         "metas_con_EA_extraido": metas_with_ea,
5563:         "metas_con_EA_extraido_pct": round(metas_with_ea_pct, 2),
5564:         "enlaces_con_logica_causal": enlaces_with_logic,
5565:         "enlaces_con_logica_causal_pct": round(enlaces_with_logic_pct, 2),
5566:         "metas_con_trazabilidad_evidencia": metas_passed_audit,
5567:         "metas_con_trazabilidad_evidencia_pct": round(metas_with_traceability_pct, 2),
5568:         "metas_con_trazabilidad_financiera": metas_with_financial,
5569:         "metas_con_trazabilidad_financiera_pct": round(metas_with_financial_pct, 2)
5570:     },
5571:     "financial_audit": {
5572:         "tablas_financieras_parseadas_exitosamente": financial_auditor.successful_parses,
5573:         "tablas_financieras_fallidas": financial_auditor.failed_parses,
5574:         "asignaciones_presupuestarias_rastreadas": len(financial_auditor.financial_data)
5575:     },
5576:     "sequence_audit": {
5577:         "alertas_secuencia_logica": len(sequence_warnings),
5578:         "detalles": sequence_warnings
5579:     },
5580:     "type_distribution": dict(type_distribution),
5581:     "rigor_distribution": dict(rigor_distribution),
5582:     "audit_summary": {
5583:         "total_audited": len(audit_results),
5584:         "passed": sum(1 for r in audit_results.values() if r['passed']),
5585:         "failed": sum(1 for r in audit_results.values() if not r['passed']),
5586:         "total_warnings": sum(len(r['warnings']) for r in audit_results.values()),
5587:         "total_errors": sum(len(r['errors']) for r in audit_results.values())
5588:     },
5589:     "quality_score": self._calculate_quality_score(
5590:         metas_with_traceability_pct,
5591:         metas_with_financial_pct,
5592:         enlaces_with_logic_pct,
5593:         metas_with_ea_pct
5594:     )
5595: }
5596:
5597: try:
5598:     with open(json_path, 'w', encoding='utf-8') as f:
5599:         json.dump(report, f, indent=2, ensure_ascii=False)
5600:     self.logger.info(f"Reporte de confianza guardado en: {json_path}")
5601: except Exception as e:
5602:     self.logger.error(f"Error guardando reporte de confianza: {e}")
5603:
5604: return json_path
5605:
5606: @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score")
5607: def _calculate_quality_score(self, traceability: float, financial: float,
5608:                             logic: float, ea: float) -> float:
5609:     """Calculate overall quality score (0-100)"""
5610:     weights = {'traceability': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "traceability_weight",
0.35), 'financial': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "financial_weight", 0.25), 'logic': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "logic_weight", 0.25), 'ea': ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine._calculate_quality_score", "ea_weight", 0.15)}

```

```
5611:         score = (traceability * weights['traceability'] +
5612:                         financial * weights['financial'] +
5613:                             logic * weights['logic'] +
5614:                               ea * weights['ea'])
5615:             return round(score, 2)
5616:
5617:     @calibrated_method("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_model_json")
5618:     def generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
5619:                                     policy_code: str) -> Path:
5620:         """Generate structured JSON export of causal model"""
5621:         json_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"
5622:
5623:         # Prepare node data
5624:         nodes_data = {}
5625:         for node_id, node in nodes.items():
5626:             node_dict = asdict(node)
5627:             # Convert NamedTuple to dict
5628:             if node.entity_activity:
5629:                 node_dict['entity_activity'] = node.entity_activity._asdict()
5630:             nodes_data[node_id] = node_dict
5631:
5632:         # Prepare edge data
5633:         edges_data = []
5634:         for source, target in graph.edges():
5635:             edge_dict = {
5636:                 'source': source,
5637:                 'target': target,
5638:                 **graph.edges[source, target]
5639:             }
5640:             edges_data.append(edge_dict)
5641:
5642:         model_data = {
5643:             "policy_code": policy_code,
5644:             "framework_version": "2." + str(ParameterLoaderV2.get("farfan_core.analysis.derek_beach.ReportingEngine.generate_causal_model_json", "framework_version", 0)),
5645:             "nodes": nodes_data,
5646:             "edges": edges_data,
5647:             "statistics": {
5648:                 "total_nodes": len(nodes_data),
5649:                 "total_edges": len(edges_data),
5650:                 "node_types": {
5651:                     node_type: sum(1 for n in nodes.values() if n.type == node_type)
5652:                     for node_type in ['programa', 'producto', 'resultado', 'impacto']
5653:                 }
5654:             }
5655:         }
5656:
5657:         try:
5658:             with open(json_path, 'w', encoding='utf-8') as f:
5659:                 json.dump(model_data, f, indent=2, ensure_ascii=False)
5660:                 self.logger.info(f"Modelo causal JSON guardado en: {json_path}")
5661:         except Exception as e:
5662:             self.logger.error(f"Error guardando modelo causal: {e}")
5663:
5664:         return json_path
5665:
```

```
5666: class CDAFFramework:
5667:     """Main orchestrator for the CDAF pipeline"""
5668:
5669:     def __init__(self, config_path: Path, output_dir: Path, log_level: str = "INFO") -> None:
5670:         self.logger = logging.getLogger(self.__class__.__name__)
5671:         self.logger.setLevel(getattr(logging, log_level.upper()))
5672:
5673:         # Initialize components
5674:         self.config = ConfigLoader(config_path)
5675:         self.output_dir = output_dir
5676:
5677:         # Initialize retry handler for external dependencies
5678:         try:
5679:             from retry_handler import DependencyType, get_retry_handler
5680:             self.retry_handler = get_retry_handler()
5681:             retry_enabled = True
5682:         except ImportError:
5683:             self.logger.warning("RetryHandler no disponible, funcionando sin retry logic")
5684:             self.retry_handler = None
5685:             retry_enabled = False
5686:
5687:         # Load spaCy model with retry logic
5688:         if retry_enabled and self.retry_handler:
5689:             @self.retry_handler.with_retry(
5690:                 DependencyType.SPACY_MODEL,
5691:                 operation_name="load_spacy_model",
5692:                 exceptions=(OSError, IOError, ImportError)
5693:             )
5694:             def load_spacy_with_retry():
5695:                 try:
5696:                     nlp = spacy.load("es_core_news_lg")
5697:                     self.logger.info("Modelo spaCy cargado: es_core_news_lg")
5698:                     return nlp
5699:                 except OSError:
5700:                     self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando es_core_news_sm...")
5701:                     nlp = spacy.load("es_core_news_sm")
5702:                     return nlp
5703:                 try:
5704:                     self.nlp = load_spacy_with_retry()
5705:                 except OSError:
5706:                     self.logger.error("No se encontró ningún modelo de spaCy en español. "
5707:                                     "Ejecute: python -m spacy download es_core_news_lg")
5708:                     sys.exit(1)
5709:             else:
5710:                 # Fallback to original logic without retry
5711:                 try:
5712:                     self.nlp = spacy.load("es_core_news_lg")
5713:                     self.logger.info("Modelo spaCy cargado: es_core_news_lg")
5714:                 except OSError:
5715:                     self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando es_core_news_sm...")
5716:                     try:
5717:                         self.nlp = spacy.load("es_core_news_sm")
5718:                     except OSError:
5719:                         self.logger.error("No se encontró ningún modelo de spaCy en español. "
5720:                                         "Ejecute: python -m spacy download es_core_news_lg")
5721:                     sys.exit(1)
```

```
5722:  
5723:     # Initialize modules (pass retry_handler to PDF processor)  
5724:     self.pdf_processor = PDFProcessor(self.config, retry_handler=self.retry_handler if retry_enabled else None)  
5725:     self.causal_extractor = CausalExtractor(self.config, self.nlp)  
5726:     self.mechanism_extractor = MechanismPartExtractor(self.config, self.nlp)  
5727:     self.bayesian_mechanism = BayesianMechanismInference(self.config, self.nlp)  
5728:     self.financial_auditor = FinancialAuditor(self.config)  
5729:     self.op_auditor = OperationalizationAuditor(self.config)  
5730:     self.inference_setup = CausalInferenceSetup(self.config)  
5731:     self.reporting_engine = ReportingEngine(self.config, output_dir)  
5732:  
5733:     # Initialize DNP validator if available  
5734:     self.dnp_validator = None  
5735:     if DNP_AVAILABLE:  
5736:         self.dnp_validator = ValidadorDNP(es_municipio_pdet=False) # Can be configured  
5737:         self.logger.info("Validador DNP inicializado")  
5738:  
5739: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework.process_document")  
5740: def process_document(self, pdf_path: Path, policy_code: str) -> bool:  
5741:     """Main processing pipeline"""  
5742:     self.logger.info(f"Iniciando procesamiento de documento: {pdf_path}")  
5743:  
5744:     try:  
5745:         # Step 1: Load and extract PDF  
5746:         if not self.pdf_processor.load_document(pdf_path):  
5747:             return False  
5748:  
5749:         text = self.pdf_processor.extract_text()  
5750:         tables = self.pdf_processor.extract_tables()  
5751:         self.pdf_processor.extract_sections()  
5752:  
5753:         # Step 2: Extract causal hierarchy  
5754:         self.logger.info("Extrayendo jerarquÃ-a causal...")  
5755:         graph = self.causal_extractor.extract_causal_hierarchy(text)  
5756:         nodes = self.causal_extractor.nodes  
5757:  
5758:         # Step 3: Extract Entity-Activity pairs  
5759:         self.logger.info("Extrayendo tuplas Entidad-Actividad...")  
5760:         for node in nodes.values():  
5761:             if node.type == 'producto':  
5762:                 ea = self.mechanism_extractor.extract_entity_activity(node.text)  
5763:                 if ea:  
5764:                     node.entity_activity = ea  
5765:                     graph.nodes[node.id]['entity_activity'] = ea._asdict()  
5766:  
5767:         # Step 4: Financial traceability  
5768:         self.logger.info("Auditando trazabilidad financiera...")  
5769:         self.financial_auditor.trace_financial_allocation(tables, nodes, graph)  
5770:  
5771:         # Step 4.5: Bayesian Mechanism Inference (AGUJA II)  
5772:         self.logger.info("Infiriendo mecanismos causales con modelo Bayesiano...")  
5773:         inferred_mechanisms = self.bayesian_mechanism.infer_mechanisms(nodes, text)  
5774:  
5775:         # Step 5: Operationalization audit  
5776:         self.logger.info("Auditando operacionalizaciÃ³n...")  
5777:         audit_results = self.op_auditor.audit_evidence_traceability(nodes)
```

```

5778:         sequence_warnings = self.op_auditor.audit_sequence_logic(graph)
5779:
5780:         # Step 5.5: Bayesian Counterfactual Audit (AGUJA III)
5781:         # Note: pdet_alignment should be calculated separately if needed via financiero_viability_tablas
5782:         # For now, using None as placeholder - can be enhanced by integrating PDETmunicipalPlanAnalyzer
5783:         self.logger.info("Ejecutando auditorÃ¡a contrafactual Bayesiana...") 
5784:         counterfactual_audit = self.op_auditor.bayesian_counterfactual_audit(nodes, graph, pdet_alignment=None)
5785:
5786:         # Step 6: Causal inference setup
5787:         self.logger.info("Preparando para inferencia causal...")
5788:         self.inference_setup.classify_goal_dynamics(nodes)
5789:         self.inference_setup.assign_probative_value(nodes)
5790:         self.inference_setup.identify_failure_points(graph, text)
5791:
5792:         # Step 7: DNP Standards Validation (if available)
5793:         if self.dnp_validator:
5794:             self.logger.info("Validando cumplimiento de estÃ¡ndares DNP...")
5795:             self._validate_dnp_compliance(nodes, graph, policy_code)
5796:
5797:         # Step 8: Generate reports
5798:         self.logger.info("Generando reportes y visualizaciones...")
5799:         self.reporting_engine.generate_causal_diagram(graph, policy_code)
5800:         self.reporting_engine.generate_accountability_matrix(graph, policy_code)
5801:         self.reporting_engine.generate_confidence_report(
5802:             nodes, graph, self.causal_extractor.causal_chains,
5803:             audit_results, self.financial_auditor, sequence_warnings, policy_code
5804:         )
5805:         self.reporting_engine.generate_causal_model_json(graph, nodes, policy_code)
5806:
5807:         # Step 8: Generate Bayesian inference reports
5808:         self.logger.info("Generando reportes de inferencia Bayesiana...")
5809:         self._generate_bayesian_reports(
5810:             inferred_mechanisms, counterfactual_audit, policy_code
5811:         )
5812:
5813:         # Step 9: Self-reflective learning from audit results (frontier paradigm)
5814:         if self.config.validated_config and self.config.validated_config.self_reflection.enable_prior_learning:
5815:             self.logger.info("Actualizando priors con retroalimentaciÃ³n del anÃ¡lisis...")
5816:             feedback_data = self._extract_feedback_from_audit(
5817:                 inferred_mechanisms, counterfactual_audit, audit_results
5818:             )
5819:             self.config.update_priors_from_feedback(feedback_data)
5820:
5821:             # HARMONIC FRONT 4: Check uncertainty reduction criterion
5822:             if hasattr(self.bayesian_mechanism, '_mean_mechanism_uncertainty'):
5823:                 uncertainty_check = self.config.check_uncertainty_reduction_criterion(
5824:                     self.bayesian_mechanism._mean_mechanism_uncertainty
5825:                 )
5826:                 self.logger.info(
5827:                     f"Uncertainty criterion check: {uncertainty_check['status']} "
5828:                     f"({{uncertainty_check['iterations_tracked']}}/10 iterations, "
5829:                     f"{{uncertainty_check['reduction_percent']:.2f}}% reduction)"
5830:                 )
5831:
5832:             self.logger.info(f"\u00c3\u2024\205 Procesamiento completado exitosamente para {policy_code}")
5833:             return True

```

```
5834:
5835:     except CDAFException as e:
5836:         # Structured error handling with custom exceptions
5837:         self.logger.error(f"Error CDAF: {e.message}")
5838:         self.logger.error(f"Detalles: {json.dumps(e.to_dict(), indent=2)}")
5839:     if not erecoverable:
5840:         raise
5841:     return False
5842: except Exception as e:
5843:     # Wrap unexpected errors in CDAFProcessingError
5844:     raise CDAFProcessingError(
5845:         "Error crÃ¡tico en el procesamiento",
5846:         details={'error': str(e), 'type': type(e).__name__},
5847:         stage="document_processing",
5848:         recoverable=False
5849:     ) from e
5850:
5851: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._extract_feedback_from_audit")
5852: def _extract_feedback_from_audit(self, inferred_mechanisms: dict[str, dict[str, Any]],
5853:                                 counterfactual_audit: dict[str, Any],
5854:                                 audit_results: dict[str, AuditResult]) -> dict[str, Any]:
5855:     """
5856:     Extract feedback data from audit results for self-reflective prior updating
5857:
5858:     This implements the frontier paradigm of learning from audit results
5859:     to improve future inference accuracy.
5860:
5861:     HARMONIC FRONT 4 ENHANCEMENT:
5862:     - Reduces mechanism_type_priors for mechanisms with implementation_failure flags
5863:     - Tracks necessity/sufficiency test failures
5864:     - Penalizes "miracle" mechanisms that fail counterfactual tests
5865:     """
5866:     feedback = {}
5867:
5868:     # Extract mechanism type frequencies from successful inferences
5869:     mechanism_frequencies = defaultdict(float)
5870:     failure_frequencies = defaultdict(float) # NEW: Track failures
5871:     total_mechanisms = 0
5872:     total_failures = 0
5873:
5874:     # Get causal implications from audit
5875:     causal_implications = counterfactual_audit.get('causal_implications', {})
5876:
5877:     for node_id, mechanism in inferred_mechanisms.items():
5878:         mechanism_type_dist = mechanism.get('mechanism_type', {})
5879:         # Weight by confidence (coherence score)
5880:         confidence = mechanism.get('coherence_score', 0.5)
5881:
5882:         # Check for implementation_failure flags in audit results
5883:         node_implications = causal_implications.get(node_id, {})
5884:         causal_effects = node_implications.get('causal_effects', {})
5885:         hasImplementationFailure = 'implementation_failure' in causal_effects
5886:
5887:         # Check necessity/sufficiency test results
5888:         necessity_test = mechanism.get('necessity_test', {})
5889:         sufficiency_test = mechanism.get('sufficiency_test', {})
```

```

5890:     failed_necessity = not necessity_test.get('is_necessary', True)
5891:     failed_sufficiency = not sufficiency_test.get('is_sufficient', True)
5892:
5893:     # If mechanism failed tests or has implementation_failure flag
5894:     if hasImplementationFailure or failed_necessity or failed_sufficiency:
5895:         total_failures += 1
5896:         # Track which mechanism types are associated with failures
5897:         for mech_type, prob in mechanism_type_dist.items():
5898:             failure_frequencies[mech_type] += prob * confidence
5899:
5900:     else:
5901:         # Only count successes for positive reinforcement
5902:         for mech_type, prob in mechanism_type_dist.items():
5903:             mechanism_frequencies[mech_type] += prob * confidence
5904:             total_mechanisms += confidence
5905:
5906:     # Normalize frequencies
5907:     if total_mechanisms > 0:
5908:         mechanism_frequencies = {
5909:             k: v / total_mechanisms
5910:             for k, v in mechanism_frequencies.items()
5911:         }
5912:         feedback['mechanism_frequencies'] = dict(mechanism_frequencies)
5913:
5914:     # NEW: Calculate penalty factors for failed mechanism types
5915:     if total_failures > 0:
5916:         failure_frequencies = {
5917:             k: v / total_failures
5918:             for k, v in failure_frequencies.items()
5919:         }
5920:         feedback['failure_frequencies'] = dict(failure_frequencies)
5921:
5922:     # Calculate penalty: reduce priors for frequently failing types
5923:     penalty_factors = {}
5924:     for mech_type, failure_freq in failure_frequencies.items():
5925:         # Higher failure frequency = stronger penalty (0.7 to 0.95) reduction
5926:         penalty_factors[mech_type] = 0.95 - (failure_freq * 0.25)
5927:     feedback['penalty_factors'] = penalty_factors
5928:
5929:     # Add audit quality metrics for future reference
5930:     feedback['audit_quality'] = {
5931:         'total_nodes audited': len(audit_results),
5932:         'passed_count': sum(1 for r in audit_results.values() if r['passed']),
5933:         'success_rate': sum(1 for r in audit_results.values() if r['passed']) / max(len(audit_results), 1),
5934:         'failure_count': total_failures, # NEW
5935:         'failure_rate': total_failures / max(len(inferred_mechanisms), 1) # NEW
5936:     }
5937:
5938:     # Track necessity/sufficiency failures for iterative validation loop
5939:     necessity_failures = sum(1 for m in inferred_mechanisms.values()
5940:                               if not m.get('necessity_test', {}).get('is_necessary', True))
5941:     sufficiency_failures = sum(1 for m in inferred_mechanisms.values()
5942:                                if not m.get('sufficiency_test', {}).get('is_sufficient', True))
5943:
5944:     feedback['test_failures'] = {
5945:         'necessity_failures': necessity_failures,
5946:         'sufficiency_failures': sufficiency_failures

```

```
5946:         }
5947:
5948:     return feedback
5949:
5950:     @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._validate_dnp_compliance")
5951:     def _validate_dnp_compliance(self, nodes: dict[str, MetaNode],
5952:                                     graph: nx.DiGraph, policy_code: str) -> None:
5953:         """
5954:             Validate DNP compliance for all nodes/projects
5955:             Generates DNP compliance report
5956:         """
5957:         if not self.dnp_validator:
5958:             return
5959:
5960:         # Build project list from nodes
5961:         proyectos = []
5962:         for node_id, node in nodes.items():
5963:             # Extract sector from responsible entity or type
5964:             sector = "general"
5965:             if node.responsible_entity:
5966:                 entity_lower = node.responsible_entity.lower()
5967:                 if "educaci" in entity_lower or "edu" in entity_lower:
5968:                     sector = "educacion"
5969:                 elif "salud" in entity_lower:
5970:                     sector = "salud"
5971:                 elif "agua" in entity_lower or "acueducto" in entity_lower:
5972:                     sector = "agua_potable_saneamiento"
5973:                 elif (
5974:                     "via" in entity_lower or "vial" in entity_lower or "transporte" in entity_lower or "infraestructura" in entity_lower):
5975:                     sector = "vias_transporte"
5976:                 elif "agr" in entity_lower or "rural" in entity_lower:
5977:                     sector = "desarrollo_agropecuario"
5978:
5979:             # Infer indicators from node type
5980:             indicadores = []
5981:             if node.type == "producto":
5982:                 # Map to MGA product indicators based on sector
5983:                 if sector == "educacion":
5984:                     indicadores = ["EDU-020", "EDU-021"]
5985:                 elif sector == "salud":
5986:                     indicadores = ["SAL-020", "SAL-021"]
5987:                 elif sector == "agua_potable_saneamiento":
5988:                     indicadores = ["APS-020", "APS-021"]
5989:             elif node.type == "resultado":
5990:                 # Map to MGA result indicators
5991:                 if sector == "educacion":
5992:                     indicadores = ["EDU-001", "EDU-002"]
5993:                 elif sector == "salud":
5994:                     indicadores = ["SAL-001", "SAL-002"]
5995:                 elif sector == "agua_potable_saneamiento":
5996:                     indicadores = ["APS-001", "APS-002"]
5997:
5998:             proyectos.append({
5999:                 "nombre": node_id,
6000:                 "sector": sector,
6001:                 "descripcion": node.text[:200] if node.text else "",
```

```
6002:         "indicadores": indicadores,
6003:         "presupuesto": node.financial_allocation or ParameterLoaderV2.get("farfan_core.analysis.derek_beach.CDAFFramework._validate_dnp_compliance", "de
fault_presupuesto", 0.0),
6004:         "es_rural": "rural" in node.text.lower() if node.text else False,
6005:         "poblacion_victimas": "v ctima" in node.text.lower() if node.text else False
6006:     ))
6007:
6008:     # Validate each project
6009:     dnp_results = []
6010:     for proyecto in proyectos:
6011:         resultado = self.dnp_validator.validar_proyecto_integral(
6012:             sector=proyecto["sector"],
6013:             descripcion=proyecto["descripcion"],
6014:             indicadores_propuestos=proyecto["indicadores"],
6015:             presupuesto=proyecto["presupuesto"],
6016:             es_rural=proyecto["es_rural"],
6017:             poblacion_victimas=proyecto["poblacion_victimas"]
6018:         )
6019:         dnp_results.append({
6020:             "proyecto": proyecto["nombre"],
6021:             "resultado": resultado
6022:         })
6023:
6024:     # Generate DNP compliance report
6025:     self._generate_dnp_report(dnp_results, policy_code)
6026:
6027:     @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_dnp_report")
6028:     def _generate_dnp_report(self, dnp_results: list[dict], policy_code: str) -> None:
6029:         """Generate comprehensive DNP compliance report"""
6030:         report_path = self.output_dir / f"{policy_code}{DNP_REPORT_SUFFIX}"
6031:
6032:         total_proyectos = len(dnp_results)
6033:         if total_proyectos == 0:
6034:             return
6035:
6036:         # Calculate aggregate statistics
6037:         proyectos_excelente = sum(1 for r in dnp_results
6038:             if r["resultado"].nivel_cumplimiento.value == "excelente")
6039:         proyectos_bueno = sum(1 for r in dnp_results
6040:             if r["resultado"].nivel_cumplimiento.value == "bueno")
6041:         proyectos_aceptable = sum(1 for r in dnp_results
6042:             if r["resultado"].nivel_cumplimiento.value == "aceptable")
6043:         proyectos_insuficiente = sum(1 for r in dnp_results
6044:             if r["resultado"].nivel_cumplimiento.value == "insuficiente")
6045:
6046:         score_promedio = sum(r["resultado"].score_total for r in dnp_results) / total_proyectos
6047:
6048:         # Build report
6049:         lines = []
6050:         lines.append("=" * 100)
6051:         lines.append("REPORTE DE CUMPLIMIENTO DE ESTÃNDARES DNP")
6052:         lines.append(f"CÃ³digo de PolÃtica: {policy_code}")
6053:         lines.append("=" * 100)
6054:         lines.append("")
6055:
6056:         lines.append("RESUMEN EJECUTIVO")
```

```

6057:     lines.append("-" * 100)
6058:     lines.append(f"Total de Proyectos/Metas Analizados: {total_proyectos}")
6059:     lines.append(f"Score Promedio de Cumplimiento: {score_promedio:.1f}/100")
6060:     lines.append("")
6061:     lines.append("DistribuciÃ³n por Nivel de Cumplimiento:")
6062:     lines.append(
6063:         f"  à\200¢ Excelente (>90%):      {proyectos_excelente:3d} ({proyectos_excelente / total_proyectos * 100:.1f}%)"
6064:     lines.append(
6065:         f"  à\200¢ Bueno (75-90%):      {proyectos_bueno:3d} ({proyectos_bueno / total_proyectos * 100:.1f}%)"
6066:     lines.append(
6067:         f"  à\200¢ Aceptable (60-75%):  {proyectos_aceptable:3d} ({proyectos_aceptable / total_proyectos * 100:.1f}%)"
6068:     lines.append(
6069:         f"  à\200¢ Insuficiente (<60%): {proyectos_insuficiente:3d} ({proyectos_insuficiente / total_proyectos * 100:.1f}%)"
6070:     lines.append("")
6071:
6072:     # Detailed validation per project
6073:     lines.append("VALIDACIÃN DETALLADA POR PROYECTO/META")
6074:     lines.append("==" * 100)
6075:
6076:     for i, result_data in enumerate(dnp_results, 1):
6077:         proyecto = result_data["proyecto"]
6078:         resultado = result_data["resultado"]
6079:
6080:         lines.append("")
6081:         lines.append(f"{i}. {proyecto}")
6082:         lines.append("-" * 100)
6083:         lines.append(
6084:             f"    Score: {resultado.score_total:.1f}/100 | Nivel: {resultado.nivel_cumplimiento.value.upper()}")
6085:
6086:         # Competencias
6087:         comp_status = "à\234\223" if resultado.cumple_competencias else "à\234\227"
6088:         lines.append(f"    Competencias Municipales: {comp_status}")
6089:         if resultado.competencias_validadas:
6090:             lines.append(f"        - Aplicables: {', '.join(resultado.competencias_validadas[:3])}")
6091:
6092:         # MGA Indicators
6093:         mga_status = "à\234\223" if resultado.cumple_mga else "à\234\227"
6094:         lines.append(f"    Indicadores MGA: {mga_status}")
6095:         if resultado.indicadores_mga_usados:
6096:             lines.append(f"        - Usados: {', '.join(resultado.indicadores_mga_usados)}")
6097:         if resultado.indicadores_mga_faltantes:
6098:             lines.append(f"        - Recomendados: {', '.join(resultado.indicadores_mga_faltantes)}")
6099:
6100:         # PDET (if applicable)
6101:         if resultado.es_municipio_pdet:
6102:             pdet_status = "à\234\223" if resultado.cumple_pdet else "à\234\227"
6103:             lines.append(f"    Lineamientos PDET: {pdet_status}")
6104:             if resultado.lineamientos_pdet_cumplidos:
6105:                 lines.append(f"        - Cumplidos: {len(resultado.lineamientos_pdet_cumplidos)}")
6106:
6107:         # Critical alerts
6108:         if resultado.alertas_criticas:
6109:             lines.append("    à\232 ALERTAS CRÃTICAS:")
6110:             for alerta in resultado.alertas_criticas:
6111:                 lines.append(f"        - {alerta}")
6112:

```

```

6113:         # Recommendations
6114:         if resultado.recomendaciones:
6115:             lines.append("  \u03b3\237\223\213 RECOMENDACIONES:")
6116:             for rec in resultado.recomendaciones[:3]: # Top 3
6117:                 lines.append(f"      - {rec}")
6118:
6119:             lines.append("")
6120:             lines.append("=" * 100)
6121:             lines.append("NORMATIVA DE REFERENCIA")
6122:             lines.append("-" * 100)
6123:             lines.append("\u00e1\u200d Competencias Municipales: Ley 136/1994, Ley 715/2001, Ley 1551/2012")
6124:             lines.append("\u00e1\u200d Indicadores MGA: DNP - Metodolog\u00e1-a General Ajustada")
6125:             lines.append("\u00e1\u200d PDET: Decreto 893/2017, Acuerdo Final de Paz")
6126:             lines.append("=" * 100)
6127:
6128:     # Write report
6129:     try:
6130:         with open(report_path, 'w', encoding='utf-8') as f:
6131:             f.write('\n'.join(lines))
6132:             self.logger.info(f"Reporte de cumplimiento DNP guardado en: {report_path}")
6133:     except Exception as e:
6134:         self.logger.error(f"Error guardando reporte DNP: {e}")
6135:
6136: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._audit_causal_coherence")
6137: def _audit_causal_coherence(self, graph: nx.DiGraph, nodes: dict[str, MetaNode]) -> dict[str, Any]:
6138: """
6139:     Audit causal coherence of the extracted model.
6140:
6141:     Args:
6142:         graph: Causal graph
6143:         nodes: Dictionary of nodes
6144:
6145:     Returns:
6146:         Dictionary with coherence audit results
6147: """
6148:     audit = {
6149:         'total_nodes': len(nodes),
6150:         'total_edges': graph.number_of_edges(),
6151:         'disconnected_nodes': [],
6152:         'cycles': [],
6153:         'coherence_score': 0.0
6154:     }
6155:
6156:     # Check for disconnected nodes
6157:     for node_id in nodes:
6158:         if graph.has_node(node_id) and graph.degree(node_id) == 0:
6159:             audit['disconnected_nodes'].append(node_id)
6160:
6161:     # Check for cycles (should not exist in causal DAG)
6162:     try:
6163:         cycles = list(nx.simple_cycles(graph))
6164:         audit['cycles'] = cycles
6165:     except:
6166:         pass
6167:
6168:     # Calculate coherence score

```

```

6169:     connected_ratio = 1.0 - (len(audit['disconnected_nodes']) / max(len(nodes), 1))
6170:     acyclic_score = 1.0 if len(audit['cycles']) == 0 else 0.5
6171:     audit['coherence_score'] = (connected_ratio + acyclic_score) / 2.0
6172:
6173:     return audit
6174:
6175: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_causal_model_json")
6176: def _generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
6177:                                 policy_code: str) -> None:
6178:     """
6179:         Generate JSON representation of causal model.
6180:
6181:     Args:
6182:         graph: Causal graph
6183:         nodes: Dictionary of nodes
6184:         policy_code: Policy code for filename
6185:     """
6186:     model = {
6187:         'policy_code': policy_code,
6188:         'nodes': [],
6189:         'edges': []
6190:     }
6191:
6192:     # Add nodes
6193:     for node_id, node in nodes.items():
6194:         model['nodes'].append({
6195:             'id': node_id,
6196:             'text': node.text,
6197:             'type': node.type,
6198:             'baseline': str(node.baseline) if node.baseline else None,
6199:             'target': str(node.target) if node.target else None
6200:         })
6201:
6202:     # Add edges
6203:     for source, target in graph.edges():
6204:         edge_data = graph.get_edge_data(source, target)
6205:         model['edges'].append({
6206:             'source': source,
6207:             'target': target,
6208:             'logic': edge_data.get('logic', 'unknown'),
6209:             'strength': edge_data.get('strength', 0.5)
6210:         })
6211:
6212:     # Write to file
6213:     output_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"
6214:     try:
6215:         with open(output_path, 'w', encoding='utf-8') as f:
6216:             json.dump(model, f, indent=2, ensure_ascii=False)
6217:             self.logger.info(f"Causal model JSON saved to: {output_path}")
6218:     except Exception as e:
6219:         self.logger.error(f"Error saving causal model JSON: {e}")
6220:
6221: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_dnp_compliance_report")
6222: def _generate_dnp_compliance_report(self, nodes: dict[str, MetaNode],
6223:                                     policy_code: str) -> dict[str, Any]:
6224:     """

```

```
6225:     Generate DNP compliance report.
6226:
6227:     Args:
6228:         nodes: Dictionary of nodes
6229:         policy_code: Policy code
6230:
6231:     Returns:
6232:         Compliance report dictionary
6233:     """
6234:     report = {
6235:         'policy_code': policy_code,
6236:         'total_products': 0,
6237:         'compliant_products': 0,
6238:         'compliance_rate': 0.0,
6239:         'gaps': []
6240:     }
6241:
6242:     # Check products for DNP compliance
6243:     for node_id, node in nodes.items():
6244:         if node.type == 'producto':
6245:             report['total_products'] += 1
6246:
6247:             # Check required fields
6248:             has_baseline = node.baseline is not None
6249:             has_target = node.target is not None
6250:             has_indicator = len(node.text) > 10 # Simple check
6251:
6252:             is_compliant = has_baseline and has_target and has_indicator
6253:
6254:             if is_compliant:
6255:                 report['compliant_products'] += 1
6256:             else:
6257:                 gaps = []
6258:                 if not has_baseline:
6259:                     gaps.append('missing_baseline')
6260:                 if not has_target:
6261:                     gaps.append('missing_target')
6262:                 if not has_indicator:
6263:                     gaps.append('missing_indicator')
6264:
6265:                 report['gaps'].append({
6266:                     'node_id': node_id,
6267:                     'issues': gaps
6268:                 })
6269:
6270:             if report['total_products'] > 0:
6271:                 report['compliance_rate'] = report['compliant_products'] / report['total_products']
6272:
6273:     return report
6274:
6275: @calibrated_method("farfan_core.analysis.derek_beach.CDAFFramework._generate_extraction_report")
6276: def _generate_extraction_report(self, nodes: dict[str, MetaNode],
6277:                                 graph: nx.DiGraph,
6278:                                 policy_code: str) -> None:
6279:     """
6280:     Generate extraction confidence report.
```

```

6281:
6282:     Args:
6283:         nodes: Dictionary of nodes
6284:         graph: Causal graph
6285:         policy_code: Policy code
6286:     """
6287:     report = {
6288:         'policy_code': policy_code,
6289:         'extraction_summary': {
6290:             'total_nodes': len(nodes),
6291:             'total_edges': graph.number_of_edges(),
6292:             'nodes_by_type': {}
6293:         },
6294:         'node_confidence': []
6295:     }
6296:
6297:     # Count nodes by type
6298:     for node in nodes.values():
6299:         node_type = node.type
6300:         report['extraction_summary']['nodes_by_type'][node_type] = \
6301:             report['extraction_summary']['nodes_by_type'].get(node_type, 0) + 1
6302:
6303:     # Add confidence scores
6304:     for node_id, node in nodes.items():
6305:         confidence = 0.8 # Refactored
6306:         if hasattr(node, 'rigor_status'):
6307:             if node.rigor_status == 'fuerte':
6308:                 confidence = 0.9 # Refactored
6309:             elif node.rigor_status == 'dÃ©bil':
6310:                 confidence = 0.6 # Refactored
6311:
6312:             report['node_confidence'].append({
6313:                 'node_id': node_id,
6314:                 'confidence': confidence
6315:             })
6316:
6317:     # Write report
6318:     output_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"
6319:     try:
6320:         with open(output_path, 'w', encoding='utf-8') as f:
6321:             json.dump(report, f, indent=2, ensure_ascii=False)
6322:             self.logger.info(f"Extraction report saved to: {output_path}")
6323:     except Exception as e:
6324:         self.logger.error(f"Error saving extraction report: {e}")
6325:
6326: # =====
6327: # AGUJA I: PRIOR ADAPTATIVO (EVIDENCIA \206\222 BAYES)
6328: # =====
6329:
6330: class BayesFactorTable:
6331:     """Tabla fija de Bayes Factors por tipo de test evidencial (Beach & Pedersen 2019)"""
6332:     FACTORS = {
6333:         'straw': (1.0, 1.5),      # STRAW_IN_WIND: Weak evidence
6334:         'hoop': (3.0, 5.0),       # HOOP TEST: Necessary but not sufficient
6335:         'smoking': (10.0, 30.0),  # SMOKING_GUN: Sufficient but not necessary
6336:         'doubly': (50.0, 100.0)   # DOUBLY_DECISIVE: Necessary AND sufficient

```

```
6337:     }
6338:
6339:     @classmethod
6340:     def get_bayes_factor(cls, test_type: str) -> float:
6341:         """Obtiene BF medio para tipo de test"""
6342:         if test_type not in cls.FACTORS:
6343:             return 1.5 # Default straw-in-wind
6344:         min_bf, max_bf = cls.FACTORS[test_type]
6345:         return (min_bf + max_bf) / 2.0
6346:
6347:     @classmethod
6348:     def get_version(cls) -> str:
6349:         """Version de tabla BF para trazabilidad"""
6350:         return "Beach2019_v1.0"
6351:
6352: class AdaptivePriorCalculator:
6353:     """
6354:     AGUJA I - Prior Adaptativo con Bayes Factor y calibraciÃ³n
6355:
6356:     PROMPT I-1: PonderaciÃ³n evidencial con BF y calibraciÃ³n
6357:     Mapea test_type\206\222BayesFactor, calcula likelihood adaptativo combinando
6358:     dominios {semantic, temporal, financial, structural} con pesos normalizados.
6359:
6360:     PROMPT I-2: Sensibilidad, OOD y ablation evidencial
6361:     Perturba cada componente ±10% y reporta \210\202p/\210\202component top-3.
6362:
6363:     PROMPT I-3: Trazabilidad y reproducibilidad
6364:     Con semilla fija, guarda bf_table_version, weights_version, snippets.
6365:
6366:     QUALITY CRITERIA:
6367:     - BrierScore ≤ 0.20) en validaciÃ³n sintÃ©tica
6368:     - ACE \210\210 [\210\2220.02), 0.02] (Average Calibration Error)
6369:     - Cobertura CI95% \210\210 [92%, 98%]
6370:     - Monotonidad: \206\221 seÃ±ales \206\222 \206\223 p_mechanism
6371:     """
6372:
6373:     def __init__(self, calibration_params: dict[str, float] | None = None) -> None:
6374:         self.logger = logging.getLogger(self.__class__.__name__)
6375:         self.bf_table = BayesFactorTable()
6376:
6377:         # Calibration params: logit\201\211(Î± + Î²Â•score)
6378:         self.calibration = calibration_params or {
6379:             'alpha': -2.0, # Intercept
6380:             'beta': 4.0 # Slope
6381:         }
6382:
6383:         # Domain weights (normalized)
6384:         self.default_domain_weights = {
6385:             'semantic': 0.35,
6386:             'temporal': 0.25,
6387:             'financial': 0.25,
6388:             'structural': 0.15
6389:         }
6390:
6391:     def calculate_likelihood_adaptativo(
6392:         self,
```

```

6393:         evidence_dict: dict[str, Any],
6394:         test_type: str = 'hoop'
6395:     ) -> dict[str, Any]:
6396:         """
6397:             PROMPT I-1: Calcula likelihood adaptativo con BF y dominios
6398:
6399:             Args:
6400:                 evidence_dict: Evidencia por caso {semantic, temporal, financial, structural}
6401:                 test_type: Tipo de test evidencial (straw, hoop, smoking, doubly)
6402:
6403:             Returns:
6404:                 Dict con p_mechanism, BF_used, domain_weights, triangulation_bonus, etc.
6405:             """
6406:             # 1. Obtener Bayes Factor para test_type
6407:             bf_used = self.bf_table.get_bayes_factor(test_type)
6408:
6409:             # 2. Extraer scores por dominio
6410:             domain_scores = {
6411:                 'semantic': evidence_dict.get('semantic', {}).get('score', 0.0),
6412:                 'temporal': evidence_dict.get('temporal', {}).get('score', 0.0),
6413:                 'financial': evidence_dict.get('financial', {}).get('score', 0.0),
6414:                 'structural': evidence_dict.get('structural', {}).get('score', 0.0)
6415:             }
6416:
6417:             # 3. Ajustar pesos si falta dominio (baja peso a 0, reparte)
6418:             adjusted_weights = self._adjust_domain_weights(domain_scores)
6419:
6420:             # 4. Calcular score combinado normalizado
6421:             combined_score = sum(
6422:                 domain_scores[domain] * adjusted_weights[domain]
6423:                 for domain in domain_scores
6424:             )
6425:
6426:             # 5. Aplicar multiplicador BF normalizado
6427:             all bfs = [np.mean(bf_range) for bf_range in self.bf_table.FACTORS.values()]
6428:             mean_bf = np.mean(all bfs)
6429:             bf_multiplier = bf_used / mean_bf
6430:             adapted_score = combined_score * bf_multiplier
6431:
6432:             # 6. Bonus de triangulaciÃ³n si >= 3 dominios activos
6433:             active_domains = sum(1 for s in domain_scores.values() if s > 0.1)
6434:             triangulation_bonus = 0.05 if active_domains >= 3 else 0.0
6435:
6436:             final_score = min(1.0, adapted_score + triangulation_bonus)
6437:
6438:             # 7. Transformar a probabilidad con logit inverso: p = 1/(1+exp(-(Î±+Î²*score)))
6439:             alpha = self.calibration['alpha']
6440:             beta = self.calibration['beta']
6441:             logit_value = alpha + beta * final_score
6442:             p_mechanism = 1.0 / (1.0 + np.exp(-logit_value))
6443:
6444:             # 8. Clip [1e-6, 1-1e-6]
6445:             p_mechanism = np.clip(p_mechanism, 1e-6, 1 - 1e-6)
6446:
6447:             return {
6448:                 'p_mechanism': float(p_mechanism),

```

```

6449:         'BF_used': bf_used,
6450:         'domain_weights': adjusted_weights,
6451:         'triangulation_bonus': triangulation_bonus,
6452:         'calibration_params': self.calibration,
6453:         'test_type': test_type,
6454:         'combined_score': combined_score,
6455:         'active_domains': active_domains
6456:     }
6457:
6458:     @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator._adjust_domain_weights")
6459:     def _adjust_domain_weights(self, domain_scores: dict[str, float]) -> dict[str, float]:
6460:         """Ajusta pesos si falta dominio: baja a 0 y reparte"""
6461:         adjusted = self.default_domain_weights.copy()
6462:
6463:         # Identificar dominios faltantes (score <= 0)
6464:         missing_domains = [d for d, s in domain_scores.items() if s <= 0]
6465:
6466:         if missing_domains:
6467:             # Bajar peso a 0 para dominios faltantes
6468:             total_missing_weight = sum(adjusted[d] for d in missing_domains)
6469:             for d in missing_domains:
6470:                 adjusted[d] = 0.0
6471:
6472:             # Repartir peso entre dominios activos
6473:             active_domains = [d for d in adjusted if adjusted[d] > 0]
6474:             if active_domains:
6475:                 bonus_per_domain = total_missing_weight / len(active_domains)
6476:                 for d in active_domains:
6477:                     adjusted[d] += bonus_per_domain
6478:
6479:             # Renormalizar para asegurar suma = 1.0
6480:             total = sum(adjusted.values())
6481:             if total > 0:
6482:                 adjusted = {k: v / total for k, v in adjusted.items()}
6483:
6484:         return adjusted
6485:
6486:     def sensitivity_analysis(
6487:         self,
6488:         evidence_dict: dict[str, Any],
6489:         test_type: str = 'hoop',
6490:         perturbation: float = 0.10
6491:     ) -> dict[str, Any]:
6492:         """
6493:             PROMPT I-2: Sensibilidad, OOD y ablation evidencial
6494:
6495:             Perturba cada componente ±10% y reporta los componentes top-3.
6496:             Ejecuta ablaciones: sólo textual, sólo financiero, sólo estructural.
6497:
6498:             CRITERIA:
6499:             - |delta_p_sensitivity|_max ≤ 0.15
6500:             - sign_concordance ≈ 2/3
6501:             - OOD_drop ≤ 0.10
6502:         """
6503:
6504:         # Baseline
6505:         baseline_result = self.calculate_likelihood_adaptativo(evidence_dict, test_type)

```

```

6505:     baseline_p = baseline_result['p_mechanism']
6506:
6507:     # 1. Sensibilidad por componente
6508:     sensitivity_map = {}
6509:     for domain in ['semantic', 'temporal', 'financial', 'structural']:
6510:         if domain in evidence_dict and isinstance(evidence_dict[domain], dict) and 'score' in evidence_dict[domain]:
6511:             # Perturbar +10%
6512:             perturbed_evidence = self._perturb_evidence(evidence_dict, domain, perturbation)
6513:             perturbed_result = self.calculate_likelihood_adaptativo(perturbed_evidence, test_type)
6514:             delta_p = perturbed_result['p_mechanism'] - baseline_p
6515:
6516:             sensitivity_map[domain] = {
6517:                 'delta_p': delta_p,
6518:                 'relative_change': delta_p / max(baseline_p, 1e-6)
6519:             }
6520:
6521:     # Top-3 por magnitud
6522:     top_3 = sorted(
6523:         sensitivity_map.items(),
6524:         key=lambda x: abs(x[1]['delta_p']),
6525:         reverse=True
6526:     )[:3]
6527:
6528:     # 2. Ablaciones: sÃ³lo un dominio
6529:     ablation_results = {}
6530:     for domain in ['semantic', 'financial', 'structural']:
6531:         ablated_evidence = {
6532:             domain: evidence_dict.get(domain, {'score': 0.0})
6533:         }
6534:         if ablated_evidence[domain].get('score', 0) > 0:
6535:             abl_result = self.calculate_likelihood_adaptativo(ablated_evidence, test_type)
6536:             ablation_results[f'only_{domain}'] = {
6537:                 'p_mechanism': abl_result['p_mechanism'],
6538:                 'sign_match': (abl_result['p_mechanism'] > 0.5) == (baseline_p > 0.5)
6539:             }
6540:
6541:     # Sign concordance
6542:     sign_concordance = sum(
6543:         1 for r in ablation_results.values() if r['sign_match']
6544:     ) / max(len(ablation_results), 1)
6545:
6546:     # 3. OOD con ruido
6547:     ood_evidence = self._add_ood_noise(evidence_dict)
6548:     ood_result = self.calculate_likelihood_adaptativo(ood_evidence, test_type)
6549:     ood_drop = abs(baseline_p - ood_result['p_mechanism'])
6550:
6551:     # 4. EvaluaciÃ³n de criterios
6552:     max_sensitivity = max((abs(item[1]['delta_p'])) for item in top_3), default=0.0
6553:     criteria_met = {
6554:         'max_sensitivity_ok': max_sensitivity <= 0.15,
6555:         'sign_concordance_ok': sign_concordance >= 2/3,
6556:         'ood_drop_ok': ood_drop <= 0.10
6557:     }
6558:
6559:     # Determinar si caso es frÃ¡gil
6560:     is_fragile = not all(criteria_met.values())

```

```

6561:
6562:     return {
6563:         'influence_top3': [(domain, data['delta_p']) for domain, data in top_3],
6564:         'delta_p_sensitivity': max_sensitivity,
6565:         'sign_concordance': sign_concordance,
6566:         'OOD_drop': ood_drop,
6567:         'ablation_results': ablation_results,
6568:         'criteria_met': criteria_met,
6569:         'is_fragile': is_fragile,
6570:         'recommendation': 'downgrade' if is_fragile else 'accept'
6571:     }
6572:
6573: def _perturb_evidence(
6574:     self,
6575:     evidence_dict: dict[str, Any],
6576:     domain: str,
6577:     perturbation: float
6578: ) -> dict[str, Any]:
6579:     """Perturba un dominio específico"""
6580:     import copy
6581:     perturbed = copy.deepcopy(evidence_dict)
6582:     if domain in perturbed and isinstance(perturbed[domain], dict) and 'score' in perturbed[domain]:
6583:         perturbed[domain]['score'] *= (1.0 + perturbation)
6584:         perturbed[domain]['score'] = min(1.0, perturbed[domain]['score'])
6585:     return perturbed
6586:
6587: @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator._add_ood_noise")
6588: def _add_ood_noise(self, evidence_dict: dict[str, Any]) -> dict[str, Any]:
6589:     """Genera set OOD con ruido semántico y tablas malformadas"""
6590:     import copy
6591:     ood = copy.deepcopy(evidence_dict)
6592:
6593:     # Agregar ruido gaussiano a todos los scores
6594:     for domain in ood:
6595:         if isinstance(ood[domain], dict) and 'score' in ood[domain]:
6596:             noise = np.random.normal(0, 0.05) # 5% noise
6597:             ood[domain]['score'] = np.clip(ood[domain]['score'] + noise, 0.0, 1.0)
6598:
6599:     return ood
6600:
6601: def generate_traceability_record(
6602:     self,
6603:     evidence_dict: dict[str, Any],
6604:     test_type: str,
6605:     result: dict[str, Any],
6606:     seed: int = 42
6607: ) -> dict[str, Any]:
6608:     """
6609:     PROMPT I-3: Trazabilidad y reproducibilidad
6610:
6611:     Con semilla fija, guarda bf_table_version, weights_version,
6612:     snippets textuales con offsets, campos financieros usados.
6613:
6614:     METRICS:
6615:     - Re-ejecución con misma semilla produce hash_result idéntico
6616:     - trace_completeness ≈ 211 ± 0.95

```

```

6617: """
6618:     # Fijar semilla para reproducibilidad
6619:     np.random.seed(seed)
6620:
6621:     # Construir evidence trace
6622:     evidence_trace = []
6623:     for domain, data in evidence_dict.items():
6624:         if isinstance(data, dict) and 'score' in data:
6625:             trace_item = {
6626:                 'source': domain,
6627:                 'line_span': data.get('line_span', 'unknown'),
6628:                 'transform_before': data.get('raw_value', None),
6629:                 'transform_after': data['score'],
6630:                 'snippet': data.get('snippet', '')[:100]  # Primeros 100 chars
6631:             }
6632:             evidence_trace.append(trace_item)
6633:
6634:     # Config hash
6635:     config_str = json.dumps({
6636:         'bf_table_version': self.bf_table.get_version(),
6637:         'calibration_params': self.calibration,
6638:         'domain_weights': self.default_domain_weights,
6639:         'test_type': test_type,
6640:         'seed': seed
6641:     }, sort_keys=True)
6642:
6643:     config_hash = hashlib.sha256(config_str.encode()).hexdigest()[:16]
6644:
6645:     # Result hash
6646:     result_str = json.dumps(result, sort_keys=True)
6647:     result_hash = hashlib.sha256(result_str.encode()).hexdigest()[:16]
6648:
6649:     # Trace completeness
6650:     factors_in_trace = len(evidence_trace)
6651:     total_factors = len([d for d in evidence_dict if isinstance(evidence_dict.get(d), dict)])
6652:     trace_completeness = factors_in_trace / max(total_factors, 1)
6653:
6654:     return {
6655:         'evidence_trace': evidence_trace,
6656:         'hash_config': config_hash,
6657:         'hash_result': result_hash,
6658:         'seed': seed,
6659:         'bf_table_version': self.bf_table.get_version(),
6660:         'weights_version': 'default_v1.0',
6661:         'trace_completeness': trace_completeness,
6662:         'reproducibility_guaranteed': trace_completeness >= 0.95
6663:     }
6664:
6665:     @calibrated_method("farfan_core.analysis.derek_beach.AdaptivePriorCalculator.validate_quality_criteria")
6666:     def validate_quality_criteria(self, validation_samples: list[dict[str, Any]]) -> dict[str, Any]:
6667:         """
6668:             Valida criterios de calidad en conjunto de validaciÃ³n sintÃ©tica
6669:
6670:             QUALITY CRITERIA:
6671:             - BrierScore <= 0.20
6672:             - ACE <= 210\210 [â\210\2220.02], 0.02)

```

```
6673:     - Cobertura CI95% à\210\210 [92%, 98%]
6674:     - Monotonidad verificada
6675:     """
6676:     predictions = []
6677:     actuals = []
6678:
6679:     for sample in validation_samples:
6680:         evidence = sample.get('evidence', {})
6681:         actual_label = sample.get('actual_label', 0.5)
6682:         test_type = sample.get('test_type', 'hoop')
6683:
6684:         result = self.calculate_likelihood_adaptativo(evidence, test_type)
6685:         predictions.append(result['p_mechanism'])
6686:         actuals.append(actual_label)
6687:
6688:     predictions = np.array(predictions)
6689:     actuals = np.array(actuals)
6690:
6691:     # 1. Brier Score
6692:     brier_score = np.mean((predictions - actuals) ** 2)
6693:     brier_ok = brier_score <= 0.20
6694:
6695:     # 2. ACE (Average Calibration Error)
6696:     # Dividir en bins
6697:     n_bins = 10
6698:     bin_boundaries = np.linspace(0, 1, n_bins + 1)
6699:     ace = 0.0 # Refactored
6700:
6701:     for i in range(n_bins):
6702:         bin_mask = (predictions >= bin_boundaries[i]) & (predictions < bin_boundaries[i + 1])
6703:         if bin_mask.sum() > 0:
6704:             bin_accuracy = actuals[bin_mask].mean()
6705:             bin_confidence = predictions[bin_mask].mean()
6706:             ace += abs(bin_accuracy - bin_confidence) / n_bins
6707:
6708:     ace_ok = -0.02 <= ace <= 0.02
6709:
6710:     # 3. Cobertura CI95%
6711:     # Simular con bootstrap
6712:     n_bootstrap = 100
6713:     coverage_count = 0
6714:
6715:     for _ in range(n_bootstrap):
6716:         idx = np.random.choice(len(predictions), size=len(predictions), replace=True)
6717:         boot_preds = predictions[idx]
6718:         boot_actuals = actuals[idx]
6719:
6720:         # Calcular CI95%
6721:         ci_low = np.percentile(boot_preds, 2.5)
6722:         ci_high = np.percentile(boot_preds, 97.5)
6723:
6724:         # Verificar si mean actual està; dentro
6725:         actual_mean = boot_actuals.mean()
6726:         if ci_low <= actual_mean <= ci_high:
6727:             coverage_count += 1
6728:
```

```

6729:     coverage = coverage_count / n_bootstrap
6730:     coverage_ok = 0.92 <= coverage <= 0.98
6731:
6732:     # 4. Monotonicidad: verificar que las señales p_mechanism
6733:     monotonicity_violations = 0
6734:
6735:     for i in range(len(validation_samples) - 1):
6736:         current_total = sum(
6737:             validation_samples[i]['evidence'].get(d, {}).get('score', 0)
6738:             for d in ['semantic', 'temporal', 'financial', 'structural']
6739:         )
6740:         next_total = sum(
6741:             validation_samples[i + 1]['evidence'].get(d, {}).get('score', 0)
6742:             for d in ['semantic', 'temporal', 'financial', 'structural']
6743:         )
6744:
6745:         if next_total > current_total and predictions[i + 1] < predictions[i]:
6746:             monotonicity_violations += 1
6747:
6748:     monotonicity_ok = monotonicity_violations == 0
6749:
6750:     return {
6751:         'brier_score': float(brier_score),
6752:         'brier_ok': brier_ok,
6753:         'ace': float(ace),
6754:         'ace_ok': ace_ok,
6755:         'ci95_coverage': float(coverage),
6756:         'coverage_ok': coverage_ok,
6757:         'monotonicity_violations': monotonicity_violations,
6758:         'monotonicity_ok': monotonicity_ok,
6759:         'all_criteria_met': brier_ok and ace_ok and coverage_ok and monotonicity_ok,
6760:         'quality_grade': 'EXCELLENT' if (brier_ok and ace_ok and coverage_ok and monotonicity_ok) else 'NEEDS_IMPROVEMENT'
6761:     }
6762:
6763: # =====
6764: # AGUJA II: MODELO GENERATIVO JERÁRQUICO
6765: # =====
6766:
6767: class HierarchicalGenerativeModel:
6768:     """
6769:     AGUJA II - Modelo Generativo Jerárquico con inferencia MCMC
6770:
6771:     PROMPT II-1: Inferencia jerárquica con incertidumbre
6772:     Estima posterior(mechanism_type, activity_sequence | obs) con MCMC.
6773:
6774:     PROMPT II-2: Posterior Predictive Checks + Ablation
6775:     Genera datos simulados desde posterior y compara con observados.
6776:
6777:     PROMPT II-3: Independencias y parsimonia
6778:     Verifica d-separaciones y calcula WAIC.
6779:
6780:     QUALITY CRITERIA:
6781:     - R-hat ≈ 1.10
6782:     - ESS ≈ 200
6783:     - entropy/entropy_max < 0.7) para certeza
6784:     - ppd_p_value ≈ 210 [0.1, 0.9]

```

```
6785:     - ï\224WAIC â\211¤ â\210\2222 para preferir jerÃ;rquico
6786:     """
6787:
6788:     def __init__(self, mechanism_priors: dict[str, float] | None = None) -> None:
6789:         self.logger = logging.getLogger(self.__class__.__name__)
6790:
6791:         # Priors dÃ©biles para mechanism_type si no se proveen
6792:         self.mechanism_priors = mechanism_priors or {
6793:             'administrativo': 0.30,
6794:             'tecnico': 0.25,
6795:             'financiero': 0.20,
6796:             'politico': 0.15,
6797:             'mixto': 0.10
6798:         }
6799:
6800:         # Validar que suman ~1.0
6801:         prior_sum = sum(self.mechanism_priors.values())
6802:         if abs(prior_sum - 1.0) > 0.01:
6803:             self.logger.warning(f"Mechanism priors sum to {prior_sum:.3f}, normalizing...")
6804:             self.mechanism_priors = {
6805:                 k: v / prior_sum for k, v in self.mechanism_priors.items()
6806:             }
6807:
6808:     def infer_mechanism_posterior(
6809:         self,
6810:         observations: dict[str, Any],
6811:         n_iter: int = 500,
6812:         burn_in: int = 100,
6813:         n_chains: int = 2
6814:     ) -> dict[str, Any]:
6815:         """
6816:             PROMPT II-1: Inferencia jerÃ;rquica con MCMC
6817:
6818:             Estima posterior(mechanism_type, activity_sequence | obs) usando MCMC.
6819:
6820:             Args:
6821:                 observations: Dict con {verbos, co_ocurrencias, coherence, structural_signals}
6822:                 n_iter: Iteraciones MCMC (â\211¥500)
6823:                 burn_in: Burn-in iterations (â\211¥100)
6824:                 n_chains: NÃºmero de cadenas para R-hat (â\211¥2)
6825:
6826:             Returns:
6827:                 Dict con type_posterior, sequence_mode, coherence_score, entropy, CI95, R-hat, ESS
6828:             """
6829:             self.logger.info(f"Starting MCMC inference: {n_iter} iter, {burn_in} burn-in, {n_chains} chains")
6830:
6831:             # Validar observaciones mÃ¬nimas
6832:             if not observations or 'coherence' not in observations:
6833:                 self.logger.warning("Missing observations, using weak priors")
6834:                 observations = observations or {}
6835:                 observations.setdefault('coherence', 0.5)
6836:
6837:             # Ejecutar mÃºltiples cadenas para diagnÃ³stico
6838:             chains = []
6839:             for chain_idx in range(n_chains):
6840:                 chain_samples = self._run_mcmc_chain(
```

```

6841:         observations, n_iter, burn_in, seed=42 + chain_idx
6842:     )
6843:     chains.append(chain_samples)
6844:     self.logger.debug(f"Chain {chain_idx + 1}/{n_chains} completed: {len(chain_samples)} samples")
6845:
6846: # Agregar samples de todas las cadenas
6847: all_samples = []
6848: for chain in chains:
6849:     all_samples.extend(chain)
6850:
6851: # 1. Type posterior (frecuencias de mechanism_type)
6852: type_counts = dict.fromkeys(self.mechanism_priors.keys(), 0)
6853: for sample in all_samples:
6854:     mtype = sample.get('mechanism_type', 'mixto')
6855:     if mtype in type_counts:
6856:         type_counts[mtype] += 1
6857:
6858: total_samples = len(all_samples)
6859: type_posterior = {
6860:     mtype: count / max(total_samples, 1)
6861:     for mtype, count in type_counts.items()
6862: }
6863:
6864: # 2. Sequence mode (secuencia más frecuente)
6865: sequence_mode = self._get_mode_sequence(all_samples)
6866:
6867: # 3. Coherence score (estadísticas)
6868: coherence_scores = [s.get('coherence', 0.5) for s in all_samples]
6869: coherence_mean = float(np.mean(coherence_scores))
6870: coherence_std = float(np.std(coherence_scores))
6871:
6872: # 4. Entropy del posterior
6873: posterior_probs = list(type_posterior.values())
6874: entropy_posterior = -sum(p * np.log(p + 1e-10) for p in posterior_probs if p > 0)
6875: max_entropy = np.log(len(self.mechanism_priors))
6876: normalized_entropy = entropy_posterior / max_entropy if max_entropy > 0 else 0.0
6877:
6878: # 5. CI95 para coherence
6879: ci95_low = float(np.percentile(coherence_scores, 2.5))
6880: ci95_high = float(np.percentile(coherence_scores, 97.5))
6881:
6882: # 6. R-hat aproximado (between-chain variance / within-chain variance)
6883: r_hat = self._calculate_r_hat(chains)
6884:
6885: # 7. ESS (Effective Sample Size)
6886: ess = self._calculate_ess(all_samples)
6887:
6888: # 8. Verificar criterios de calidad
6889: is_uncertain = normalized_entropy > 0.7
6890: criteria_met = {
6891:     'r_hat_ok': r_hat <= 1.10,
6892:     'ess_ok': ess >= 200,
6893:     'entropy_ok': not is_uncertain
6894: }
6895:
6896: # Warning si alta incertidumbre

```

```
6897:     warning = None
6898:     if is_uncertain:
6899:         warning = f"HIGH_UNCERTAINTY: entropy/entropy_max = {normalized_entropy:.3f} > 0.7)"
6900:         self.logger.warning(warning)
6901:
6902:     return {
6903:         'type_posterior': type_posterior,
6904:         'sequence_mode': sequence_mode,
6905:         'coherence_score': coherence_mean,
6906:         'coherence_std': coherence_std,
6907:         'entropy_posterior': float(entropy_posterior),
6908:         'normalized_entropy': float(normalized_entropy),
6909:         'CI95': (ci95_low, ci95_high),
6910:         'CI95_width': ci95_high - ci95_low,
6911:         'R_hat': float(r_hat),
6912:         'ESS': float(ess),
6913:         'n_samples': total_samples,
6914:         'is_uncertain': is_uncertain,
6915:         'criteria_met': criteria_met,
6916:         'warning': warning
6917:     }
6918:
6919: def _run_mcmc_chain(
6920:     self,
6921:     observations: dict[str, Any],
6922:     n_iter: int,
6923:     burn_in: int,
6924:     seed: int
6925: ) -> list[dict[str, Any]]:
6926:     """Ejecuta una cadena MCMC con Metropolis-Hastings"""
6927:     np.random.seed(seed)
6928:     samples = []
6929:
6930:     # Estado inicial: sample desde prior
6931:     current_type = np.random.choice(
6932:         list(self.mechanism_priors.keys()),
6933:         p=list(self.mechanism_priors.values())
6934:     )
6935:     current_coherence = observations.get('coherence', 0.5)
6936:
6937:     for i in range(n_iter):
6938:         # Proporcionar nuevo mechanism_type
6939:         proposed_type = np.random.choice(list(self.mechanism_priors.keys()))
6940:
6941:         # Calcular likelihood ratio
6942:         current_likelihood = self._calculate_likelihood(current_type, observations)
6943:         proposed_likelihood = self._calculate_likelihood(proposed_type, observations)
6944:
6945:         # Prior ratio
6946:         prior_ratio = self.mechanism_priors[proposed_type] / max(self.mechanism_priors[current_type], 1e-10)
6947:
6948:         # Acceptance probability (Metropolis-Hastings)
6949:         likelihood_ratio = proposed_likelihood / max(current_likelihood, 1e-10)
6950:         acceptance_prob = min(1.0, likelihood_ratio * prior_ratio)
6951:
6952:         # Accept/reject
```

```

6953:         if np.random.random() < acceptance_prob:
6954:             current_type = proposed_type
6955:
6956:             # Simular coherence con ruido
6957:             simulated_coherence = current_coherence + np.random.normal(0, 0.05)
6958:             simulated_coherence = np.clip(simulated_coherence, 0.0, 1.0)
6959:
6960:             # Almacenar sample (despuÃ±s de burn-in)
6961:             if i >= burn_in:
6962:                 sample = {
6963:                     'mechanism_type': current_type,
6964:                     'coherence': float(simulated_coherence),
6965:                     'iteration': i - burn_in,
6966:                     'chain_seed': seed
6967:                 }
6968:                 samples.append(sample)
6969:
6970:             return samples
6971:
6972: def _calculate_likelihood(
6973:     self,
6974:     mechanism_type: str,
6975:     observations: dict[str, Any]
6976: ) -> float:
6977:     """Calcula likelihood de observations dado mechanism_type"""
6978:     # Likelihood basado en coherence y structural signals
6979:     coherence = observations.get('coherence', 0.5)
6980:     structural_signals = observations.get('structural_signals', {})
6981:
6982:     # Base likelihood desde prior
6983:     prior = self.mechanism_priors.get(mechanism_type, 0.1)
6984:
6985:     # Ajuste por coherence (mayor coherence \206\222 mayor likelihood)
6986:     coherence_factor = 1.0 + coherence
6987:
6988:     # Ajuste por seÃ±ales estructurales especÃ-ficas del tipo
6989:     structural_match = 0.0 # Refactored
6990:     if mechanism_type == 'administrativo' and structural_signals.get('admin_keywords', 0) > 0:
6991:         structural_match = 0.2 # Refactored
6992:     elif mechanism_type == 'financiero' and structural_signals.get('budget_data', 0) > 0:
6993:         structural_match = 0.3 # Refactored
6994:     elif mechanism_type == 'tecnico' and structural_signals.get('technical_terms', 0) > 0:
6995:         structural_match = 0.25 # Refactored
6996:
6997:     likelihood = prior * coherence_factor * (1.0 + structural_match)
6998:     return likelihood
6999:
7000: @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._get_mode_sequence")
7001: def _get_mode_sequence(self, samples: list[dict[str, Any]]) -> str:
7002:     """Obtiene secuencia modal (tipo mÃ;s frecuente)"""
7003:     type_counts = {}
7004:     for s in samples:
7005:         mtype = s.get('mechanism_type', 'mixto')
7006:         type_counts[mtype] = type_counts.get(mtype, 0) + 1
7007:
7008:     if type_counts:

```

```

7009:         return max(type_counts.items(), key=lambda x: x[1])[0]
7010:     return 'mixto'
7011:
7012:     @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_r_hat")
7013:     def _calculate_r_hat(self, chains: list[list[dict[str, Any]]]) -> float:
7014:         """Calcula Gelman-Rubin R-hat para diagnÃ³stico de convergencia"""
7015:         if len(chains) < 2:
7016:             return 1.0
7017:
7018:         # Extraer coherence de cada cadena
7019:         chain_means = []
7020:         chain_vars = []
7021:
7022:         for chain in chains:
7023:             coherences = [s.get('coherence', 0.5) for s in chain]
7024:             if len(coherences) > 0:
7025:                 chain_means.append(np.mean(coherences))
7026:                 chain_vars.append(np.var(coherences, ddof=1))
7027:
7028:             if len(chain_means) < 2:
7029:                 return 1.0
7030:
7031:         # Between-chain variance (B)
7032:         n = len(chains[0])  # samples per chain
7033:         B = np.var(chain_means, ddof=1) * n
7034:
7035:         # Within-chain variance (W)
7036:         W = np.mean(chain_vars)
7037:
7038:         # R-hat estimator
7039:         if W > 0:
7040:             var_plus = ((n - 1) / n) * W + (1 / n) * B
7041:             r_hat = np.sqrt(var_plus / W)
7042:         else:
7043:             r_hat = 1.0 # Refactored
7044:
7045:         return float(r_hat)
7046:
7047:     @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess")
7048:     def _calculate_ess(self, samples: list[dict[str, Any]]) -> float:
7049:         """Calcula Effective Sample Size (simplificado)"""
7050:         n = len(samples)
7051:
7052:         # Estimar autocorrelaciÃ³n
7053:         coherences = np.array([s.get('coherence', 0.5) for s in samples])
7054:
7055:         if len(coherences) < 2:
7056:             return n
7057:
7058:         # Lag-1 autocorrelation
7059:         mean_coh = np.mean(coherences)
7060:         var_coh = np.var(coherences)
7061:
7062:         if var_coh > 0:
7063:             lag1_autocorr = np.mean(
7064:                 (coherences[:-1] - mean_coh) * (coherences[1:] - mean_coh)

```

```
7065:         ) / var_coh
7066:     else:
7067:         lag1_autocorr = 0.0 # Refactored
7068:
7069:         # ESS approximation
7070:         ess = n / (1 + 2 * max(0, lag1_autocorr))
7071:     return float(ess)
7072:
7073: def posterior_predictive_check(
7074:     self,
7075:     posterior_samples: list[dict[str, Any]],
7076:     observed_data: dict[str, Any]
7077: ) -> dict[str, Any]:
7078:     """
7079:     PROMPT II-2: Posterior Predictive Checks + Ablation
7080:
7081:     Genera datos simulados desde posterior y compara con observados.
7082:     Realiza ablation de pasos de secuencia.
7083:
7084:     Args:
7085:         posterior_samples: Samples del posterior MCMC
7086:         observed_data: Datos observados reales
7087:
7088:     Returns:
7089:         Dict con ppd_p_value, distance_metric, ablation_curve, criteria_met
7090:     """
7091:     self.logger.info("Running posterior predictive checks...")
7092:
7093:     # 1. Generar datos predictivos desde posterior
7094:     n_ppd_samples = min(100, len(posterior_samples))
7095:     ppd_samples = []
7096:
7097:     for _i in range(n_ppd_samples):
7098:         sample_idx = np.random.randint(0, len(posterior_samples))
7099:         posterior_sample = posterior_samples[sample_idx]
7100:
7101:         # Simular coherence desde distribuciÃ³n posterior
7102:         simulated_coherence = posterior_sample.get('coherence', 0.5) + np.random.normal(0, 0.05)
7103:         simulated_coherence = np.clip(simulated_coherence, 0.0, 1.0)
7104:         ppd_samples.append(simulated_coherence)
7105:
7106:     ppd_samples = np.array(ppd_samples)
7107:
7108:     # 2. Comparar con observado usando KS test
7109:     observed_coherence = observed_data.get('coherence', 0.5)
7110:
7111:     # KS test: comparar distribuciÃ³n PPD con punto observado
7112:     from scipy.stats import kstest
7113:     ks_stat, ppd_p_value = kstest(ppd_samples, lambda x: 0 if x < observed_coherence else 1)
7114:     ppd_p_value = float(ppd_p_value)
7115:
7116:     # 3. Ablation de secuencia
7117:     ablation_curve = self._ablation_analysis(posterior_samples, observed_data)
7118:
7119:     # 4. Verificar criterios
7120:     ppd_ok = 0.1 <= ppd_p_value <= 0.9
```

```

7121:     ablation_ok = all(delta >= -0.05 for delta in ablation_curve.values())  # Tolerancia -5%
7122:
7123:     criteria_met = {
7124:         'ppd_p_value_ok': ppd_ok,
7125:         'ablation_ok': ablation_ok
7126:     }
7127:
7128:     # RecomendaciÃ³n
7129:     if ppd_ok and ablation_ok:
7130:         recommendation = 'accept'
7131:     else:
7132:         recommendation = 'rebaja_posterior'
7133:     self.logger.warning(f"PPC failed: ppd_p={ppd_p_value:.3f}, ablation_ok={ablation_ok}")
7134:
7135:     return {
7136:         'ppd_p_value': ppd_p_value,
7137:         'ppd_samples_mean': float(np.mean(ppd_samples)),
7138:         'ppd_samples_std': float(np.std(ppd_samples)),
7139:         'distance_metric': 'KS',
7140:         'ks_statistic': float(ks_stat),
7141:         'ablation_curve': ablation_curve,
7142:         'criteria_met': criteria_met,
7143:         'recommendation': recommendation
7144:     }
7145:
7146: def _ablation_analysis(
7147:     self,
7148:     posterior_samples: list[dict[str, Any]],
7149:     observed_data: dict[str, Any]
7150: ) -> dict[str, float]:
7151:     """Mide caÃ±da en coherence al quitar pasos de secuencia"""
7152:     baseline_coherence = np.mean([s.get('coherence', 0.5) for s in posterior_samples])
7153:
7154:     # Simular ablaciÃ³n de pasos clave
7155:     # En prÃ¡ctica real, esto requerirÃ¡ re-ejecutar modelo sin ciertos steps
7156:     ablation_deltas = {
7157:         'remove_step_diagnostic': baseline_coherence - (baseline_coherence * 0.95),  # -5%
7158:         'remove_step_planning': baseline_coherence - (baseline_coherence * 0.85),  # -15%
7159:         'remove_step_execution': baseline_coherence - (baseline_coherence * 0.90),  # -10%
7160:         'remove_step_monitoring': baseline_coherence - (baseline_coherence * 0.97)  # -3%
7161:     }
7162:
7163:     return ablation_deltas
7164:
7165: def verify_conditional_independence(
7166:     self,
7167:     dag: nx.DiGraph,
7168:     independence_tests: list[tuple[str, str, list[str]]] | None = None
7169: ) -> dict[str, Any]:
7170:     """
7171:     PROMPT II-3: Independencias y parsimonia
7172:
7173:     Verifica d-separaciones implicadas por el DAG.
7174:     Calcula f\224WAIC entre modelo jerÃ;rquico vs. nulo.
7175:
7176:     Args:

```

```
7177:         dag: NetworkX DiGraph del modelo causal
7178:         independence_tests: Lista de tuplas (X, Y, Z) para test X ⊥\!\!\!⊥ Y | Z
7179:
7180:     Returns:
7181:         Dict con independence_tests, delta_waic, model_preference, criteria_met
7182:         """
7183:         self.logger.info("Verifying conditional independencies...")
7184:
7185:         # 1. Tests de independencia (d-separaciÃ³n)
7186:         test_results = []
7187:
7188:         if independence_tests is None:
7189:             # Generar tests automÃ;ticamente si no se proveen
7190:             independence_tests = self._generate_independence_tests(dag)
7191:
7192:         for x, y, z_set in independence_tests:
7193:             try:
7194:                 # Verificar d-separaciÃ³n en DAG
7195:                 is_independent = nx.d_separated(dag, {x}, {y}, set(z_set))
7196:                 test_results.append({
7197:                     'test': f'{x} ⊥\!\!\!⊥ {y} | {{{', '.join(z_set)}}}',
7198:                     'x': x,
7199:                     'y': y,
7200:                     'z': z_set,
7201:                     'passed': is_independent
7202:                 })
7203:             except Exception as e:
7204:                 self.logger.warning(f"Independence test failed: {x} ⊥\!\!\!⊥ {y} | {z_set} - {e}")
7205:                 test_results.append({
7206:                     'test': f'{x} ⊥\!\!\!⊥ {y} | {{{', '.join(z_set)}}}',
7207:                     'x': x,
7208:                     'y': y,
7209:                     'z': z_set,
7210:                     'passed': False,
7211:                     'error': str(e)
7212:                 })
7213:
7214:         tests_passed = sum(1 for t in test_results if t['passed'])
7215:
7216:         # 2. Calcular f\WAIC (simplificado)
7217:         # En prÃ;ctica real: usar librerÃ;a como arviz para WAIC calculation
7218:         delta_waic = self._calculate_waic_difference(dag)
7219:
7220:         # 3. Verificar criterios
7221:         independence_ok = tests_passed >= 2
7222:         waic_ok = delta_waic <= -2.0
7223:
7224:         # 4. Preferencia de modelo
7225:         if independence_ok and waic_ok:
7226:             model_preference = 'hierarchical'
7227:         elif not waic_ok:
7228:             model_preference = 'inconclusive'
7229:         else:
7230:             model_preference = 'null'
7231:
7232:         criteria_met = {
```

```

7233:         'independence_ok': independence_ok,
7234:         'waic_ok': waic_ok
7235:     }
7236:
7237:     return {
7238:         'independence_tests': test_results,
7239:         'tests_passed': tests_passed,
7240:         'tests_total': len(test_results),
7241:         'delta_waic': float(delta_waic),
7242:         'model_preference': model_preference,
7243:         'criteria_met': criteria_met
7244:     }
7245:
7246:     def _generate_independence_tests(
7247:         self,
7248:         dag: nx.DiGraph,
7249:         n_tests: int = 3
7250:     ) -> list[tuple[str, str, list[str]]]:
7251:         """Genera tests de independencia automÁticamente desde DAG"""
7252:         tests = []
7253:         nodes = list(dag.nodes())
7254:
7255:         if len(nodes) < 3:
7256:             return tests
7257:
7258:         # Generar tests de forma heurÁstica
7259:         for _ in range(min(n_tests, len(nodes) - 2)):
7260:             # Seleccionar nodos aleatorios
7261:             x, y = np.random.choice(nodes, size=2, replace=False)
7262:
7263:             # Z: padres comunes o mediadores
7264:             z_candidates = set(dag.predecessors(x)) | set(dag.predecessors(y))
7265:             z_set = list(z_candidates)[:2] # MÁximo 2 nodos en conditioning set
7266:
7267:             if x != y:
7268:                 tests.append((x, y, z_set))
7269:
7270:         return tests
7271:
7272:     @calibrated_method("farfan_core.analysis.derek_beach.HierarchicalGenerativeModel._calculate_waic_difference")
7273:     def _calculate_waic_difference(self, dag: nx.DiGraph) -> float:
7274:         """
7275:             Calcula  $\Delta WAIC = WAIC_{hierarchical} - WAIC_{null}$  (simplificado)
7276:
7277:             En producciÃ³n: usar arviz.waic() con trace real de PyMC/Stan
7278:         """
7279:         # HeurÃstica: modelos jerÃ;quicos con mÃ;s estructura (edges) son preferidos
7280:         n_edges = dag.number_of_edges()
7281:         dag.number_of_nodes()
7282:
7283:         # PenalizaciÃ³n por complejidad
7284:         complexity_penalty = n_edges * 0.5
7285:
7286:         # WAIC aproximado
7287:         waic_hierarchical = -50.0 - n_edges * 2 # Mejor fit con mÃ;s estructura
7288:         waic_null = -45.0 # Modelo nulo sin estructura

```

```
7289:  
7290:     delta_waic = waic_hierarchical - waic_null + complexity_penalty  
7291:  
7292:     return delta_waic  
7293:  
7294: # =====  
7295: # AGUJA III: AUDITOR CONTRAFACTUAL BAYESIANO  
7296: # =====  
7297:  
7298: class BayesianCounterfactualAuditor:  
7299:     """  
7300:     AGUJA III - Auditor Contrafactual con SCM y do-calculus  
7301:  
7302:     PROMPT III-1: ConstrucciÃ³n de SCM y queries gemelas  
7303:     Construye SCM={DAG, f_i} y responde omission_impact, sufficiency_test, necessity_test.  
7304:  
7305:     PROMPT III-2: Riesgo sistÃ©mico y priorizaciÃ³n  
7306:     Agrega riesgos, propaga incertidumbre, calcula priority.  
7307:  
7308:     PROMPT III-3: RefutaciÃ³n, negativos y cordura do()  
7309:     Ejecuta controles negativos, pruebas placebo, sanity checks.  
7310:  
7311:     QUALITY CRITERIA:  
7312:     - Consistencia de signos factual/contrafactual  
7313:     - effect_stability: f\224effect ~ 0.15) al variar priors Â±10%  
7314:     - negative_controls: mediana |efecto| ~ 0.05  
7315:     - sanity_violations: 0  
7316:     """  
7317:  
7318:     def __init__(self) -> None:  
7319:         self.logger = logging.getLogger(self.__class__.__name__)  
7320:         self.scm: dict[str, Any] | None = None  
7321:  
7322:     def construct_scm(  
7323:         self,  
7324:         dag: nx.DiGraph,  
7325:         structural_equations: dict[str, callable] | None = None  
7326:     ) -> dict[str, Any]:  
7327:         """  
7328:             PROMPT III-1: ConstrucciÃ³n de SCM  
7329:  
7330:             Construye SCM = {DAG, f_i} desde grafo y ecuaciones estructurales.  
7331:  
7332:             Args:  
7333:                 dag: NetworkX DiGraph (debe ser acÃ¡clico)  
7334:                 structural_equations: Dict {node: function} para f_i  
7335:  
7336:             Returns:  
7337:                 SCM con DAG validado y funciones estructurales  
7338:  
7339:             Raises:  
7340:                 ValueError: Si DAG no es acÃ¡clico  
7341:                 """  
7342:                 self.logger.info(f"Constructing SCM with {dag.number_of_nodes()} nodes, {dag.number_of_edges()} edges")  
7343:  
7344:                 # 1. Validar que DAG es acÃ¡clico
```

```

7345:         if not nx.is_directed_acyclic_graph(dag):
7346:             raise ValueError("DAG must be acyclic for SCM construction. Use cycle detection first.")
7347:
7348:     # 2. Crear ecuaciones por defecto si no se proveen
7349:     if structural_equations is None:
7350:         structural_equations = self._create_default_equations(dag)
7351:         self.logger.info(f"Created {len(structural_equations)} default structural equations")
7352:
7353:     # 3. Construir SCM
7354:     scm = {
7355:         'dag': dag,
7356:         'equations': structural_equations,
7357:         'nodes': list(dag.nodes()),
7358:         'edges': list(dag.edges()),
7359:         'topological_order': list(nx.topological_sort(dag))
7360:     }
7361:
7362:     self.scm = scm
7363:     self.logger.info("â\234\223 SCM constructed successfully")
7364:     return scm
7365:
7366: @calibrated_method("farfan_core.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations")
7367: def _create_default_equations(self, dag: nx.DiGraph) -> dict[str, callable]:
7368:     """Crea ecuaciones estructurales lineales por defecto"""
7369:     equations = {}
7370:
7371:     for node in dag.nodes():
7372:         parents = list(dag.predecessors(node))
7373:
7374:         if not parents:
7375:             # Nodo raÃ±o: variable exÃ³gena U
7376:             def root_eq(noise=0.0, node_name=node):
7377:                 return 0.5 + noise # Prior neutral + ruido
7378:             equations[node] = root_eq
7379:         else:
7380:             # Nodo con padres: funciÃ³n lineal
7381:             def child_eq(parent_values, noise=0.0, node_name=node, n_parents=len(parents)):
7382:                 if isinstance(parent_values, dict):
7383:                     return sum(parent_values.values()) / max(n_parents, 1) + noise
7384:                 return 0.5 + noise
7385:             equations[node] = child_eq
7386:
7387:     return equations
7388:
7389: def counterfactual_query(
7390:     self,
7391:     intervention: dict[str, float],
7392:     target: str,
7393:     evidence: dict[str, float] | None = None
7394: ) -> dict[str, Any]:
7395:     """
7396:     PROMPT III-1: Queries gemelas (omission, sufficiency, necessity)
7397:
7398:     EvalÃ³a:
7399:     - Factual: P(Y | evidence)
7400:     - Counterfactual: P(Y | do(X=x), evidence)

```

```

7401:     - Causal effect, sufficiency, necessity
7402:
7403:     Args:
7404:         intervention: {nodo: valor} para do(.) operation
7405:         target: Nodo objetivo Y
7406:         evidence: Evidencia observada (opcional)
7407:
7408:     Returns:
7409:         Dict con p_factual, p_counterfactual, causal_effect, is_sufficient, is_necessary
7410:         """
7411:     if self.scm is None:
7412:         raise ValueError("SCM must be constructed first. Call construct_scm().")
7413:
7414:     evidence = evidence or {}
7415:
7416:     self.logger.debug(f"Counterfactual query: intervention={intervention}, target={target}")
7417:
7418:     # 1. Factual: P(Y | evidence)
7419:     p_factual = self._evaluate_factual(target, evidence)
7420:
7421:     # 2. Counterfactual: P(Y | do(X=x), evidence)
7422:     p_counterfactual = self._evaluate_counterfactual(target, intervention, evidence)
7423:
7424:     # 3. Causal effect
7425:     causal_effect = p_counterfactual - p_factual
7426:
7427:     # 4. Sufficiency test: ¿do(X=1) → Y=1?
7428:     intervention_node = list(intervention.keys())[0] if intervention else None
7429:     if intervention_node:
7430:         p_y_given_do_x1 = self._evaluate_counterfactual(target, {intervention_node: 1.0}, {})
7431:         is_sufficient = p_y_given_do_x1 > 0.7
7432:     else:
7433:         is_sufficient = False
7434:
7435:     # 5. Necessity test: ¿do(X=0) → Y=0?
7436:     if intervention_node:
7437:         p_y_given_do_x0 = self._evaluate_counterfactual(target, {intervention_node: 0.0}, {})
7438:         is_necessary = p_y_given_do_x0 < 0.3
7439:     else:
7440:         is_necessary = False
7441:
7442:     # 6. Consistencia de signos
7443:     signs_consistent = (
7444:         (causal_effect >= 0 and p_counterfactual >= p_factual) or
7445:         (causal_effect < 0 and p_counterfactual < p_factual)
7446:     )
7447:
7448:     # 7. Effect stability
7449:     stability = self._test_effect_stability(intervention, target, evidence)
7450:
7451:     return {
7452:         'p_factual': float(np.clip(p_factual, 0.0, 1.0)),
7453:         'p_counterfactual': float(np.clip(p_counterfactual, 0.0, 1.0)),
7454:         'causal_effect': float(causal_effect),
7455:         'is_sufficient': is_sufficient,
7456:         'is_necessary': is_necessary,

```

```
7457:         'signs_consistent': signs_consistent,
7458:         'effect_stability': float(stability),
7459:         'effect_stable': stability <= 0.15
7460:     }
7461:
7462:     def _evaluate_factual(
7463:         self,
7464:         target: str,
7465:         evidence: dict[str, float]
7466:     ) -> float:
7467:         """Evalúa P(target | evidence) propagando hacia adelante en DAG"""
7468:         if target in evidence:
7469:             return evidence[target]
7470:
7471:         dag = self.scm['dag']
7472:         equations = self.scm['equations']
7473:         topological_order = self.scm['topological_order']
7474:
7475:         # Evaluar nodos en orden topológico
7476:         computed_values = evidence.copy()
7477:
7478:         for node in topological_order:
7479:             if node in computed_values:
7480:                 continue
7481:
7482:             parents = list(dag.predecessors(node))
7483:
7484:             if not parents:
7485:                 # Nodo raíz
7486:                 computed_values[node] = equations[node](noise=0.0)
7487:             else:
7488:                 # Evaluar padres primero
7489:                 parent_values = {}
7490:                 for parent in parents:
7491:                     if parent not in computed_values:
7492:                         computed_values[parent] = self._evaluate_factual(parent, evidence)
7493:                         parent_values[parent] = computed_values[parent]
7494:
7495:                 # Aplicar ecuación estructural
7496:                 try:
7497:                     computed_values[node] = equations[node](parent_values, noise=0.0)
7498:                 except:
7499:                     # Fallback
7500:                     computed_values[node] = sum(parent_values.values()) / max(len(parent_values), 1)
7501:
7502:         return float(np.clip(computed_values.get(target, 0.5), 0.0, 1.0))
7503:
7504:     def _evaluate_counterfactual(
7505:         self,
7506:         target: str,
7507:         intervention: dict[str, float],
7508:         evidence: dict[str, float]
7509:     ) -> float:
7510:         """Evalúa P(target | do(intervention), evidence) con DAG mutilado"""
7511:         # Crear DAG mutilado: quitar aristas hacia nodos intervenidos
7512:         dag_mutilated = self.scm['dag'].copy()
```

```
7513:  
7514:     for node in intervention:  
7515:         in_edges = list(dag_mutilated.in_edges(node))  
7516:         dag_mutilated.remove_edges_from(in_edges)  
7517:  
7518:     # Guardar SCM original  
7519:     original_scm = self.scm.copy()  
7520:  
7521:     # Crear SCM mutilado temporalmente  
7522:     self.scm = {  
7523:         'dag': dag_mutilated,  
7524:         'equations': self.scm['equations'],  
7525:         'nodes': self.scm['nodes'],  
7526:         'edges': list(dag_mutilated.edges()),  
7527:         'topological_order': list(nx.topological_sort(dag_mutilated))  
7528:     }  
7529:  
7530:     # Combinar evidence con intervention (intervention tiene prioridad)  
7531:     combined_evidence = {**evidence, **intervention}  
7532:  
7533:     # Evaluar en SCM mutilado  
7534:     result = self._evaluate_factual(target, combined_evidence)  
7535:  
7536:     # Restaurar SCM original  
7537:     self.scm = original_scm  
7538:  
7539:     return result  
7540:  
7541: def _test_effect_stability(  
7542:     self,  
7543:     intervention: dict[str, float],  
7544:     target: str,  
7545:     evidence: dict[str, float] | None,  
7546:     n_perturbations: int = 5  
7547: ) -> float:  
7548:     """Testa estabilidad al variar priors/ecuaciones ±10%"""  
7549:     evidence = evidence or {}  
7550:  
7551:     # Efecto baseline  
7552:     baseline_result = self.counterfactual_query(intervention, target, evidence)  
7553:     baseline_effect = baseline_result['causal_effect']  
7554:  
7555:     # Perturbar y medir variaciÃ³n  
7556:     perturbed_effects = []  
7557:  
7558:     for _ in range(n_perturbations):  
7559:         perturbation_factor = np.random.uniform(0.9, 1.1)  # ±10%  
7560:  
7561:         # Perturbar valores de evidencia  
7562:         perturbed_evidence = {  
7563:             k: v * perturbation_factor for k, v in evidence.items()  
7564:         }  
7565:  
7566:         # Re-evaluar  
7567:         try:  
7568:             result = self.counterfactual_query(intervention, target, perturbed_evidence)
```

```

7569:         perturbed_effects.append(result['causal_effect'])
7570:     except:
7571:         perturbed_effects.append(baseline_effect)
7572:
7573:     # MÁxima variación
7574:     max_variation = max(abs(e - baseline_effect) for e in perturbed_effects) if perturbed_effects else 0.0
7575:
7576:     return max_variation
7577:
7578: def aggregate_risk_and_prioritize(
7579:     self,
7580:     omission_score: float,
7581:     insufficiency_score: float,
7582:     unnecessary_score: float,
7583:     causal_effect: float,
7584:     feasibility: float = 0.8,
7585:     cost: float = 1.0
7586: ) -> dict[str, Any]:
7587: """
7588: PROMPT III-2: Riesgo sistémico y priorización con incertidumbre
7589:
7590: Fórmulas:
7591: - risk = 0.50 * omission + 0.35 * insufficiency + 0.15 * unnecessary
7592: - priority = |effect| * feasibility / (cost + μ * (1 / 210 / 222 uncertainty))
7593:
7594: Args:
7595:     omission_score: Riesgo de omisión de mecanismo [0,1]
7596:     insufficiency_score: Insuficiencia del mecanismo [0,1]
7597:     unnecessary_score: Mecanismo innecesario [0,1]
7598:     causal_effect: Efecto causal estimado
7599:     feasibility: Factibilidad de intervención [0,1]
7600:     cost: Costo relativo (>0)
7601:
7602: Returns:
7603:     Dict con risk_score, success_probability, priority, recommendations
7604: """
7605: # 1. Componentes de riesgo
7606: risk_components = {
7607:     'omission': float(np.clip(omission_score, 0.0, 1.0)),
7608:     'insufficiency': float(np.clip(insufficiency_score, 0.0, 1.0)),
7609:     'unnecessary': float(np.clip(unnecessary_score, 0.0, 1.0))
7610: }
7611:
7612: # 2. Riesgo agregado
7613: risk_score = (
7614:     0.50 * risk_components['omission'] +
7615:     0.35 * risk_components['insufficiency'] +
7616:     0.15 * risk_components['unnecessary']
7617: )
7618: risk_score = float(np.clip(risk_score, 0.0, 1.0))
7619:
7620: # 3. Success probability con incertidumbre
7621: success_mean = 1.0 - risk_score
7622:
7623: # Incertidumbre: mayor riesgo → mayor uncertainty
7624: success_std = 0.05 + 0.10 * risk_score # Entre 5% y 15%

```

```
7625:  
7626:      # CI95 para success  
7627:      ci95_low = max(0.0, success_mean - 1.96 * success_std)  
7628:      ci95_high = min(1.0, success_mean + 1.96 * success_std)  
7629:  
7630:      success_probability = {  
7631:          'mean': float(success_mean),  
7632:          'std': float(success_std),  
7633:          'CI95': (float(ci95_low), float(ci95_high))  
7634:      }  
7635:  
7636:      # 4. Prioridad  
7637:      uncertainty = success_std  
7638:      epsilon = 1e-6  
7639:  
7640:      priority = (  
7641:          abs(causal_effect) *  
7642:          feasibility /  
7643:          (cost + epsilon) *  
7644:          (1.0 - uncertainty)  
7645:      )  
7646:      priority = float(priority)  
7647:  
7648:      # 5. Recomendaciones ordenadas  
7649:      recommendations = []  
7650:  
7651:      if risk_score > 0.7:  
7652:          recommendations.append("CRITICAL_RISK: Immediate intervention required")  
7653:      elif risk_score > 0.4:  
7654:          recommendations.append("MEDIUM_RISK: Close monitoring required")  
7655:      else:  
7656:          recommendations.append("LOW_RISK: Routine surveillance")  
7657:  
7658:      if risk_components['omission'] > 0.6:  
7659:          recommendations.append("HIGH_OMISSION_RISK: Key mechanism may be missing")  
7660:  
7661:      if risk_components['insufficiency'] > 0.5:  
7662:          recommendations.append("INSUFFICIENCY_DETECTED: Mechanism alone insufficient")  
7663:  
7664:      if priority > 0.5:  
7665:          recommendations.append("HIGH_PRIORITY: Optimal intervention candidate")  
7666:      elif priority < 0.2:  
7667:          recommendations.append("LOW_PRIORITY: Consider alternative interventions")  
7668:  
7669:      # 6. Verificar criterios de calidad  
7670:      ci95_valid = 0.0 <= ci95_low <= ci95_high <= 1.0  
7671:      priority_monotonic = priority >= 0  
7672:      risk_in_range = 0.0 <= risk_score <= 1.0  
7673:  
7674:      criteria_met = {  
7675:          'ci95_valid': ci95_valid,  
7676:          'priority_monotonic': priority_monotonic,  
7677:          'risk_in_range': risk_in_range  
7678:      }  
7679:  
7680:      return {
```

```

7681:         'risk_components': risk_components,
7682:         'risk_score': risk_score,
7683:         'success_probability': success_probability,
7684:         'priority': priority,
7685:         'recommendations': sorted(recommendations, reverse=True),
7686:         'criteria_met': criteria_met
7687:     }
7688:
7689:     def reputation_and_sanity_checks(
7690:         self,
7691:         dag: nx.DiGraph,
7692:         target: str,
7693:         treatment: str,
7694:         confounders: list[str] | None = None
7695:     ) -> dict[str, Any]:
7696:         """
7697:             PROMPT III-3: RefutaciÃ³n, negativos y cordura do(..)
7698:
7699:             Ejecuta:
7700:             1. Controles negativos: nodos irrelevantes |efecto| > 0.05
7701:             2. Pruebas placebo: permuta edges no causales
7702:             3. Sanity checks: aÃ±adir cofactores no reduce P(Y|do(X=1))
7703:
7704:             Args:
7705:                 dag: Grafo causal
7706:                 target: Nodo objetivo Y
7707:                 treatment: Nodo de tratamiento X
7708:                 confounders: Lista de cofactores
7709:
7710:             Returns:
7711:                 Dict con negative_controls, placebo_effect, sanity_violations, recommendation
7712:             """
7713:             confounders = confounders or []
7714:
7715:             self.logger.info("Running reputation and sanity checks...")
7716:
7717:             # 1. CONTROLES NEGATIVOS: nodos irrelevantes
7718:             irrelevant_nodes = [
7719:                 n for n in dag.nodes()
7720:                 if n not in (target, treatment) and not nx.has_path(dag, n, target)
7721:             ]
7722:
7723:             negative_effects = []
7724:             for node in irrelevant_nodes[:5]: # MÃ¡ximo 5 controles
7725:                 try:
7726:                     intervention = {node: 1.0}
7727:                     result = self.counterfactual_query(intervention, target, {})
7728:                     effect = abs(result['causal_effect'])
7729:                     negative_effects.append(effect)
7730:                 except Exception as e:
7731:                     self.logger.warning(f"Negative control failed for {node}: {e}")
7732:
7733:             median_negative_effect = float(np.median(negative_effects)) if negative_effects else 0.0
7734:             negative_controls_ok = median_negative_effect <= 0.05
7735:
7736:             # 2. PRUEBA PLACEBO: permuta edges no causales

```

```

7737:     placebo_dag = dag.copy()
7738:     non_causal_edges = [
7739:         (u, v) for u, v in dag.edges()
7740:         if u != treatment and v != target
7741:     ]
7742:
7743:     placebo_effect = 0.0 # Refactored
7744:     if non_causal_edges:
7745:         # Permutar una arista
7746:         edge_to_remove = non_causal_edges[0]
7747:         placebo_dag.remove_edge(*edge_to_remove)
7748:
7749:         # Medir efecto en DAG permutado
7750:         scm_backup = self.scm
7751:         try:
7752:             self.construct_scm(placebo_dag)
7753:             result = self.counterfactual_query({treatment: 1.0}, target, {})
7754:             placebo_effect = abs(result['causal_effect'])
7755:         except Exception as e:
7756:             self.logger.warning(f"Placebo test failed: {e}")
7757:         finally:
7758:             self.scm = scm_backup
7759:
7760:     placebo_ok = placebo_effect <= 0.05
7761:
7762:     # 3. SANITY CHECKS: añadir cofactores activos no debe reducir P(Y|do(X=1))
7763:     sanity_violations = []
7764:
7765:     # Baseline: do(X=1)
7766:     try:
7767:         baseline_result = self.counterfactual_query({treatment: 1.0}, target, {})
7768:         baseline_p = baseline_result['p_counterfactual']
7769:
7770:         # Con cofactores
7771:         for confounder in confounders[:2]: # Mximo 2
7772:             if confounder in dag.nodes():
7773:                 result_with_conf = self.counterfactual_query(
7774:                     {treatment: 1.0},
7775:                     target,
7776:                     {confounder: 1.0}
7777:                 )
7778:                 p_with_conf = result_with_conf['p_counterfactual']
7779:
7780:                 # Verificar que no reduce significativamente
7781:                 if p_with_conf < baseline_p - 0.10:
7782:                     sanity_violations.append({
7783:                         'confounder': confounder,
7784:                         'baseline_p': float(baseline_p),
7785:                         'p_with_confounder': float(p_with_conf),
7786:                         'violation': f"Adding {confounder} reduced P(Y|do(X)) by {baseline_p - p_with_conf:.3f}"
7787:                     })
7788:             except Exception as e:
7789:                 self.logger.error(f"Sanity checks failed: {e}")
7790:
7791:             sanity_ok = len(sanity_violations) == 0
7792:

```

```
7793:     # 4. DECISIÃN FINAL
7794:     all_checks_passed = negative_controls_ok and placebo_ok and sanity_ok
7795:
7796:     if not all_checks_passed:
7797:         recommendation = "DEGRADE_ALL: Require DAG revision - observaciÃ³n prioritaria"
7798:         self.logger.error(recommendation)
7799:     else:
7800:         recommendation = "ACCEPT: All refutation tests passed"
7801:         self.logger.info(recommendation)
7802:
7803:     return {
7804:         'negative_controls': {
7805:             'effects': [float(e) for e in negative_effects],
7806:             'median': median_negative_effect,
7807:             'passed': negative_controls_ok,
7808:             'criterion': '\u00b10.05'
7809:         },
7810:         'placebo_effect': {
7811:             'effect': float(placebo_effect),
7812:             'passed': placebo_ok,
7813:             'criterion': '\u00b1210 0'
7814:         },
7815:         'sanity_violations': sanity_violations,
7816:         'sanity_passed': sanity_ok,
7817:         'all_checks_passed': all_checks_passed,
7818:         'recommendation': recommendation
7819:     }
7820:
7821: def main() -> int:
7822:     """CLI entry point"""
7823:     parser = argparse.ArgumentParser(
7824:         description="CDAF v2.0 - Framework de DeconstrucciÃ³n y AuditorÃ‐a Causal",
7825:         formatter_class=argparse.RawDescriptionHelpFormatter,
7826:         epilog="""
7827: Ejemplo de uso:
7828:     python cdaf_framework.py documento.pdf --output-dir resultados/ --policy-code P1
7829:
7830: ConfiguraciÃ³n:
7831:     El framework busca config.yaml en el directorio actual.
7832:     Use --config-file para especificar una ruta alternativa.
7833: """
7834: )
7835:
7836:     parser.add_argument(
7837:         "pdf_path",
7838:         type=Path,
7839:         help="Ruta al archivo PDF del Plan de Desarrollo Territorial"
7840:     )
7841:
7842:     parser.add_argument(
7843:         "--output-dir",
7844:         type=Path,
7845:         default=Path("resultados_analisis"),
7846:         help="Directorio de salida para los artefactos (default: resultados_analisis/)"
7847:     )
7848:
```

```
7849:     parser.add_argument(
7850:         "--policy-code",
7851:         type=str,
7852:         required=True,
7853:         help="Código de política para nombrar los artefactos (ej: P1, PDT_2024)"
7854:     )
7855:
7856:     parser.add_argument(
7857:         "--config-file",
7858:         type=Path,
7859:         default=Path(DEFAULT_CONFIG_FILE),
7860:         help=f"Ruta al archivo de configuración YAML (default: {DEFAULT_CONFIG_FILE})"
7861:     )
7862:
7863:     parser.add_argument(
7864:         "--log-level",
7865:         choices=["DEBUG", "INFO", "WARNING", "ERROR"],
7866:         default="INFO",
7867:         help="Nivel de logging (default: INFO)"
7868:     )
7869:
7870:     parser.add_argument(
7871:         "--pdet",
7872:         action="store_true",
7873:         help="Indica si el municipio es PDET (activa validación especial)"
7874:     )
7875:
7876:     args = parser.parse_args()
7877:
7878:     # Validate inputs
7879:     if not args.pdf_path.exists():
7880:         print(f"ERROR: Archivo PDF no encontrado: {args.pdf_path}")
7881:         return 1
7882:
7883:     # Initialize framework
7884:     try:
7885:         framework = CDAFFramework(args.config_file, args.output_dir, args.log_level)
7886:
7887:         # Configure PDET if specified
7888:         if args.pdet and framework.dnp_validator:
7889:             framework.dnp_validator.es_municipio_pdet = True
7890:             framework.logger.info("Modo PDET activado - Validación especial habilitada")
7891:     except Exception as e:
7892:         print(f"ERROR: No se pudo inicializar el framework: {e}")
7893:         return 1
7894:
7895:     # Process document
7896:     success = framework.process_document(args.pdf_path, args.policy_code)
7897:
7898:     return 0 if success else 1
7899:
7900: # =====
7901: # PRODUCER CLASS - Registry Exposure
7902: # =====
7903:
7904: class DerekBeachProducer:
```

```
7905: """
7906: Producer wrapper for Derek Beach causal analysis with registry exposure
7907:
7908: Provides public API methods for orchestrator integration without exposing
7909: internal implementation details or summarization logic.
7910:
7911: Version: 1.0).0
7912: Producer Type: Causal Mechanism Analysis
7913: """
7914:
7915: def __init__(self) -> None:
7916:     """Initialize producer"""
7917:     self.logger = logging.getLogger(self.__class__.__name__)
7918:     self.logger.info("DerekBeachProducer initialized")
7919:
7920: # =====
7921: # EVIDENTIAL TESTS API
7922: # =====
7923:
7924: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.classify_test_type")
7925: def classify_test_type(self, necessity: float, sufficiency: float) -> TestType:
7926:     """Classify evidential test type based on necessity and sufficiency"""
7927:     return BeachEvidentialTest.classify_test(necessity, sufficiency)
7928:
7929: def apply_test_logic(
7930:     self,
7931:     test_type: TestType,
7932:     evidence_found: bool,
7933:     prior: float,
7934:     bayes_factor: float
7935: ) -> tuple[float, str]:
7936:     """Apply Beach test-specific logic to Bayesian updating"""
7937:     return BeachEvidentialTest.apply_test_logic(
7938:         test_type, evidence_found, prior, bayes_factor
7939:     )
7940:
7941: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_hoop_test")
7942: def is_hoop_test(self, test_type: TestType) -> bool:
7943:     """Check if test is hoop test"""
7944:     return test_type == "hoop_test"
7945:
7946: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_smoking_gun")
7947: def is_smoking_gun(self, test_type: TestType) -> bool:
7948:     """Check if test is smoking gun"""
7949:     return test_type == "smoking_gun"
7950:
7951: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_doubly_decisional")
7952: def is_doubly_decisional(self, test_type: TestType) -> bool:
7953:     """Check if test is doubly decisional"""
7954:     return test_type == "doubly_decisional"
7955:
7956: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_straw_in_wind")
7957: def is_straw_in_wind(self, test_type: TestType) -> bool:
7958:     """Check if test is straw in wind"""
7959:     return test_type == "straw_in_wind"
7960:
```

```
7961: # =====
7962: # HIERARCHICAL GENERATIVE MODEL API
7963: # =====
7964:
7965: def create_hierarchical_model(
7966:     self,
7967:     mechanism_priors: dict[str, float] | None = None
7968: ) -> HierarchicalGenerativeModel:
7969:     """Create hierarchical generative model"""
7970:     return HierarchicalGenerativeModel(mechanism_priors)
7971:
7972: def infer_mechanism_posterior(
7973:     self,
7974:     model: HierarchicalGenerativeModel,
7975:     observations: dict[str, Any],
7976:     n_iter: int = 500,
7977:     burn_in: int = 100,
7978:     n_chains: int = 2
7979: ) -> dict[str, Any]:
7980:     """Infer mechanism posterior using MCMC"""
7981:     return model.infer_mechanism_posterior(
7982:         observations, n_iter, burn_in, n_chains
7983:     )
7984:
7985: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_type_posterior")
7986: def get_type_posterior(self, inference: dict[str, Any]) -> dict[str, float]:
7987:     """Extract type posterior from inference"""
7988:     return inference.get("type_posterior", {})
7989:
7990: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_sequence_mode")
7991: def get_sequence_mode(self, inference: dict[str, Any]) -> str:
7992:     """Extract sequence mode from inference"""
7993:     return inference.get("sequence_mode", "")
7994:
7995: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_coherence_score")
7996: def get_coherence_score(self, inference: dict[str, Any]) -> float:
7997:     """Extract coherence score from inference"""
7998:     return inference.get("coherence_score", 0.0)
7999:
8000: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_r_hat")
8001: def get_r_hat(self, inference: dict[str, Any]) -> float:
8002:     """Extract R-hat convergence diagnostic"""
8003:     return inference.get("R_hat", 1.0)
8004:
8005: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ess")
8006: def get_ess(self, inference: dict[str, Any]) -> float:
8007:     """Extract effective sample size"""
8008:     return inference.get("ESS", 0.0)
8009:
8010: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_inference_uncertain")
8011: def is_inference_uncertain(self, inference: dict[str, Any]) -> bool:
8012:     """Check if inference has high uncertainty"""
8013:     return inference.get("is_uncertain", False)
8014:
8015: # =====
8016: # POSTERIOR PREDICTIVE CHECKS API
```

```
8017:     # =====
8018:
8019:     def posterior_predictive_check(
8020:         self,
8021:         model: HierarchicalGenerativeModel,
8022:         posterior_samples: list[dict[str, Any]],
8023:         observed_data: dict[str, Any]
8024:     ) -> dict[str, Any]:
8025:         """Run posterior predictive checks"""
8026:         return model.posterior_predictive_check(posterior_samples, observed_data)
8027:
8028:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ppd_p_value")
8029:     def get_ppd_p_value(self, ppc: dict[str, Any]) -> float:
8030:         """Extract posterior predictive p-value"""
8031:         return ppc.get("ppd_p_value", 0.0)
8032:
8033:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ablation_curve")
8034:     def get_ablation_curve(self, ppc: dict[str, Any]) -> dict[str, float]:
8035:         """Extract ablation curve from PPC"""
8036:         return ppc.get("ablation_curve", {})
8037:
8038:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_ppc_recommendation")
8039:     def get_ppc_recommendation(self, ppc: dict[str, Any]) -> str:
8040:         """Extract recommendation from PPC"""
8041:         return ppc.get("recommendation", "")
8042:
8043:     # =====
8044:     # CONDITIONAL INDEPENDENCE API
8045:     # =====
8046:
8047:     def verify_conditional_independence(
8048:         self,
8049:         model: HierarchicalGenerativeModel,
8050:         dag: nx.DiGraph,
8051:         independence_tests: list[tuple[str, str, list[str]]] | None = None
8052:     ) -> dict[str, Any]:
8053:         """Verify conditional independencies in DAG"""
8054:         return model.verify_conditional_independence(dag, independence_tests)
8055:
8056:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_independence_tests")
8057:     def get_independence_tests(self, verification: dict[str, Any]) -> list[dict[str, Any]]:
8058:         """Extract independence tests from verification"""
8059:         return verification.get("independence_tests", [])
8060:
8061:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_delta_waic")
8062:     def get_delta_waic(self, verification: dict[str, Any]) -> float:
8063:         """Extract delta WAIC from verification"""
8064:         return verification.get("delta_waic", 0.0)
8065:
8066:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_model_preference")
8067:     def get_model_preference(self, verification: dict[str, Any]) -> str:
8068:         """Extract model preference from verification"""
8069:         return verification.get("model_preference", "inconclusive")
8070:
8071:     # =====
8072:     # COUNTERFACTUAL AUDITOR API
```

```
8073: # =====
8074:
8075: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.create_auditor")
8076: def create_auditor(self) -> BayesianCounterfactualAuditor:
8077:     """Create Bayesian counterfactual auditor"""
8078:     return BayesianCounterfactualAuditor()
8079:
8080: def construct_scm(
8081:     self,
8082:     auditor: BayesianCounterfactualAuditor,
8083:     dag: nx.DiGraph,
8084:     structural_equations: dict[str, callable] | None = None
8085: ) -> dict[str, Any]:
8086:     """Construct structural causal model"""
8087:     return auditor.construct_scm(dag, structural_equations)
8088:
8089: def counterfactual_query(
8090:     self,
8091:     auditor: BayesianCounterfactualAuditor,
8092:     intervention: dict[str, float],
8093:     target: str,
8094:     evidence: dict[str, float] | None = None
8095: ) -> dict[str, Any]:
8096:     """Execute counterfactual query"""
8097:     return auditor.counterfactual_query(intervention, target, evidence)
8098:
8099: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_causal_effect")
8100: def get_causal_effect(self, query: dict[str, Any]) -> float:
8101:     """Extract causal effect from query"""
8102:     return query.get("causal_effect", 0.0)
8103:
8104: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_sufficient")
8105: def is_sufficient(self, query: dict[str, Any]) -> bool:
8106:     """Check if mechanism is sufficient"""
8107:     return query.get("is_sufficient", False)
8108:
8109: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_necessary")
8110: def is_necessary(self, query: dict[str, Any]) -> bool:
8111:     """Check if mechanism is necessary"""
8112:     return query.get("is_necessary", False)
8113:
8114: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.is_effect_stable")
8115: def is_effect_stable(self, query: dict[str, Any]) -> bool:
8116:     """Check if effect is stable"""
8117:     return query.get("effect_stable", False)
8118:
8119: # =====
8120: # RISK AGGREGATION API
8121: # =====
8122:
8123: def aggregate_risk(
8124:     self,
8125:     auditor: BayesianCounterfactualAuditor,
8126:     omission_score: float,
8127:     insufficiency_score: float,
8128:     unreversability_score: float,
```

```
8129:         causal_effect: float,
8130:         feasibility: float = 0.8,
8131:         cost: float = 1.0
8132:     ) -> dict[str, Any]:
8133:         """Aggregate risk and calculate priority"""
8134:         return auditor.aggregate_risk_and_prioritize(
8135:             omission_score,
8136:             insufficiency_score,
8137:             unreversability_score,
8138:             causal_effect,
8139:             feasibility,
8140:             cost
8141:         )
8142:
8143: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_risk_score")
8144: def get_risk_score(self, aggregation: dict[str, Any]) -> float:
8145:     """Extract risk score from aggregation"""
8146:     return aggregation.get("risk_score", 0.0)
8147:
8148: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_success_probability")
8149: def get_success_probability(self, aggregation: dict[str, Any]) -> dict[str, float]:
8150:     """Extract success probability from aggregation"""
8151:     return aggregation.get("success_probability", {})
8152:
8153: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_priority")
8154: def get_priority(self, aggregation: dict[str, Any]) -> float:
8155:     """Extract priority from aggregation"""
8156:     return aggregation.get("priority", 0.0)
8157:
8158: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_recommendations")
8159: def get_recommendations(self, aggregation: dict[str, Any]) -> list[str]:
8160:     """Extract recommendations from aggregation"""
8161:     return aggregation.get("recommendations", [])
8162:
8163: # =====
8164: # REFUTATION API
8165: # =====
8166:
8167: def refutation_checks(
8168:     self,
8169:     auditor: BayesianCounterfactualAuditor,
8170:     dag: nx.DiGraph,
8171:     target: str,
8172:     treatment: str,
8173:     confounders: list[str] | None = None
8174: ) -> dict[str, Any]:
8175:     """Execute refutation and sanity checks"""
8176:     return auditor.refutation_and_sanity_checks(
8177:         dag, target, treatment, confounders
8178:     )
8179:
8180: @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_negative_controls")
8181: def get_negative_controls(self, refutation: dict[str, Any]) -> dict[str, Any]:
8182:     """Extract negative controls from refutation"""
8183:     return refutation.get("negative_controls", {})
8184:
```

```
8185:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_placebo_effect")
8186:     def get_placebo_effect(self, refutation: dict[str, Any]) -> dict[str, Any]:
8187:         """Extract placebo effect from refutation"""
8188:         return refutation.get("placebo_effect", {})
8189:
8190:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_sanity_violations")
8191:     def get_sanityViolations(self, refutation: dict[str, Any]) -> list[dict[str, Any]]:
8192:         """Extract sanity violations from refutation"""
8193:         return refutation.get("sanity_violations", [])
8194:
8195:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.all_checks_passed")
8196:     def all_checks_passed(self, refutation: dict[str, Any]) -> bool:
8197:         """Check if all refutation checks passed"""
8198:         return refutation.get("all_checks_passed", False)
8199:
8200:     @calibrated_method("farfan_core.analysis.derek_beach.DerekBeachProducer.get_refutation_recommendation")
8201:     def get_refutation_recommendation(self, refutation: dict[str, Any]) -> str:
8202:         """Extract recommendation from refutation"""
8203:         return refutation.get("recommendation", "")
8204:
8205:
8206: =====
8207: =====
8208: FILE: src/farfan_pipeline/analysis/enhance_recommendation_rules.py
8209: =====
8210:
8211: #!/usr/bin/env python3
8212: """
8213: Script to enhance recommendation rules with 7 advanced features:
8214: 1. Template parameterization
8215: 2. Rule execution logic
8216: 3. Measurable indicators
8217: 4. Unambiguous time horizons
8218: 5. Testable verification
8219: 6. Cost tracking
8220: 7. Authority mapping
8221: """
8222:
8223: import copy
8224: import json
8225: from pathlib import Path
8226: from typing import Any
8227: from farfan_pipeline.core.calibration.decorators import calibrated_method
8228:
8229:
8230: def enhance_template(rule: dict[str, Any]) -> dict[str, Any]:
8231:     """
8232:         Feature 1: Eliminate Hardcoded Template Strings
8233:         Replace with template_id and template_params
8234:     """
8235:     template = rule.get('template', {})
8236:     rule_id = rule.get('rule_id', '')
8237:     level = rule.get('level', '')
8238:
8239:     # Extract parameters from template strings
8240:     template_params = {}
```

```
8241:
8242:     if level == 'MICRO':
8243:         when = rule.get('when', {})
8244:         template_params = {
8245:             'pa_id': when.get('pa_id', 'PA01'),
8246:             'dim_id': when.get('dim_id', 'DIM01'),
8247:             'question_id': 'Q001' # Would be derived from context
8248:         }
8249:     elif level == 'MESO':
8250:         when = rule.get('when', {})
8251:         template_params = {
8252:             'cluster_id': when.get('cluster_id', 'CL01')
8253:         }
8254:
8255:     # Create enhanced template with ID
8256:     enhanced_template = copy.deepcopy(template)
8257:     enhanced_template['template_id'] = f"TPL-{rule_id}"
8258:     enhanced_template['template_params'] = template_params
8259:
8260:     return enhanced_template
8261:
8262: def add_execution_logic(rule: dict[str, Any]) -> dict[str, Any]:
8263:     """
8264:         Feature 2: Add Rule Execution Logic
8265:     """
8266:     when = rule.get('when', {})
8267:     level = rule.get('level', '')
8268:
8269:     # Build trigger condition string
8270:     trigger_parts = []
8271:     if level == 'MICRO':
8272:         pa_id = when.get('pa_id', '')
8273:         dim_id = when.get('dim_id', '')
8274:         score_lt = when.get('score_lt', 1.65)
8275:         trigger_parts.append(f"score < {score_lt}")
8276:         trigger_parts.append(f"pa_id = '{pa_id}'")
8277:         trigger_parts.append(f"dim_id = '{dim_id}'")
8278:     elif level == 'MESO':
8279:         cluster_id = when.get('cluster_id', '')
8280:         score_band = when.get('score_band', '')
8281:         if score_band:
8282:             trigger_parts.append(f"score_band = '{score_band}'")
8283:             if cluster_id:
8284:                 trigger_parts.append(f"cluster_id = '{cluster_id}'")
8285:     elif level == 'MACRO':
8286:         macro_band = when.get('macro_band', '')
8287:         if macro_band:
8288:             trigger_parts.append(f"macro_band = '{macro_band}'")
8289:
8290:     trigger_condition = " AND ".join(trigger_parts) if trigger_parts else "true"
8291:
8292:     return {
8293:         "trigger_condition": trigger_condition,
8294:         "blocking": False,
8295:         "auto_apply": False,
8296:         "requires_approval": True,
```

```
8297:         "approval_roles": ["Secretaría de Planeación", "Secretaría de Hacienda"]
8298:     }
8299:
8300: def enhance_indicator(indicator: dict[str, Any], rule_id: str, level: str) -> dict[str, Any]:
8301:     """
8302:     Feature 3: Make Indicators Measurable
8303:     """
8304:     enhanced = copy.deepcopy(indicator)
8305:
8306:     # Add formula based on indicator type
8307:     indicator.get('name', '')
8308:     if 'proporción' in indicator.get('unit', ''):
8309:         enhanced['formula'] = 'COUNT(compliant_items) / COUNT(total_items)'
8310:         enhanced['acceptable_range'] = [0.6, 1.0]
8311:     elif 'porcentaje' in indicator.get('unit', ''):
8312:         enhanced['formula'] = '(achieved / target) * 100'
8313:         enhanced['acceptable_range'] = [60.0, 100.0]
8314:     else:
8315:         enhanced['formula'] = 'SUM(verified_artifacts)'
8316:         enhanced['acceptable_range'] = [indicator.get('target', 1) * 0.7, indicator.get('target', 1)]
8317:
8318:     # Add measurement metadata
8319:     enhanced['baseline_measurement_date'] = '2024-01-01'
8320:     enhanced['measurement_frequency'] = 'mensual'
8321:     enhanced['data_source'] = 'Sistema de Seguimiento de Planes (SSP)'
8322:     enhanced['data_source_query'] = f"SELECT COUNT(*) FROM indicators WHERE indicator_id = '{rule_id}-IND'"
8323:     enhanced['responsible_measurement'] = 'Oficina de Planeación Municipal'
8324:     enhanced['escalation_if_below'] = enhanced['acceptable_range'][0]
8325:
8326:     return enhanced
8327:
8328: def enhance_horizon(horizon: dict[str, str], rule_id: str) -> dict[str, Any]:
8329:     """
8330:     Feature 4: Define Unambiguous Time Horizons
8331:     """
8332:     # Map T0, T1, T2, T3 to actual durations
8333:     duration_map = {
8334:         'T0': 0,
8335:         'T1': 6,    # 6 months
8336:         'T2': 12,   # 12 months
8337:         'T3': 24    # 24 months
8338:     }
8339:
8340:     start = horizon.get('start', 'T0')
8341:     end = horizon.get('end', 'T1')
8342:
8343:     duration_months = duration_map.get(end, 6) - duration_map.get(start, 0)
8344:
8345:     # Create milestones
8346:     milestones = []
8347:     if duration_months >= 6:
8348:         milestones.append({
8349:             "name": "Inicio de implementación",
8350:             "offset_months": 1,
8351:             "deliverables": ["Plan de trabajo aprobado"],
8352:             "verification_required": True
```

```

8353:         })
8354:     if duration_months >= 12:
8355:         milestones.append({
8356:             "name": "RevisiÃ³n intermedia",
8357:             "offset_months": duration_months // 2,
8358:             "deliverables": ["Informe de avance"],
8359:             "verification_required": True
8360:         })
8361:     milestones.append({
8362:         "name": "Entrega final",
8363:         "offset_months": duration_months,
8364:         "deliverables": ["Todos los productos esperados"],
8365:         "verification_required": True
8366:     })
8367:
8368:     return {
8369:         "start": start,
8370:         "end": end,
8371:         "start_type": "plan_approval_date",
8372:         "duration_months": duration_months,
8373:         "milestones": milestones,
8374:         "dependencies": [],
8375:         "critical_path": duration_months <= 6
8376:     }
8377:
8378: def enhance_verification(verification: list[str], rule_id: str) -> list[dict[str, Any]]:
8379: """
8380: Feature 5: Make Verification Testable
8381: """
8382: enhanced_verifications = []
8383:
8384: for idx, artifact_text in enumerate(verification, 1):
8385:     # Determine artifact type
8386:     artifact_type = "DOCUMENT"
8387:     if any(word in artifact_text.lower() for word in ['sistema', 'repositorio', 'registro', 'base de datos']):
8388:         artifact_type = "SYSTEM_STATE"
8389:
8390:     # Create structured verification
8391:     ver_obj = {
8392:         "id": f"VER-{rule_id}-{idx:03d}",
8393:         "type": artifact_type,
8394:         "artifact": artifact_text,
8395:         "format": "PDF" if artifact_type == "DOCUMENT" else "DATABASE_QUERY",
8396:         "required_sections": ["Objetivo", "Alcance", "Resultados"] if artifact_type == "DOCUMENT" else [],
8397:         "approval_required": True,
8398:         "approver": "SecretarÃ‐a de PlaneaciÃ³n",
8399:         "due_date": "T1",
8400:         "automated_check": artifact_type == "SYSTEM_STATE"
8401:     }
8402:
8403:     # Add validation for system states
8404:     if artifact_type == "SYSTEM_STATE":
8405:         ver_obj["validation_query"] = f"SELECT COUNT(*) FROM artifacts WHERE artifact_id = '{ver_obj['id']}'"
8406:         ver_obj["pass_condition"] = "COUNT(*) >= 1"
8407:
8408:     enhanced_verifications.append(ver_obj)

```

```
8409:  
8410:     return enhanced_verifications  
8411:  
8412: def add_budget(rule: dict[str, Any]) -> dict[str, Any]:  
8413:     """  
8414:         Feature 6: Integrate Cost Tracking  
8415:         """  
8416:     level = rule.get('level', '')  
8417:  
8418:     # Estimate costs based on level and complexity  
8419:     if level == 'MICRO':  
8420:         base_cost = 45_000_000 # COP  
8421:     elif level == 'MESO':  
8422:         base_cost = 150_000_000  
8423:     elif level == 'MACRO':  
8424:         base_cost = 500_000_000  
8425:     else:  
8426:         base_cost = 50_000_000  
8427:  
8428:     # Cost breakdown  
8429:     personal_cost = int(base_cost * 0.55)  
8430:     consultancy_cost = int(base_cost * 0.30)  
8431:     technology_cost = int(base_cost * 0.15)  
8432:  
8433:     return {  
8434:         "estimated_cost_cop": base_cost,  
8435:         "cost_breakdown": {  
8436:             "personal": personal_cost,  
8437:             "consultancy": consultancy_cost,  
8438:             "technology": technology_cost  
8439:         },  
8440:         "funding_sources": [  
8441:             {  
8442:                 "source": "SGP - Sistema General de Participaciones",  
8443:                 "amount": int(base_cost * 0.60),  
8444:                 "confirmed": False  
8445:             },  
8446:             {  
8447:                 "source": "Recursos Propios",  
8448:                 "amount": int(base_cost * 0.40),  
8449:                 "confirmed": False  
8450:             }  
8451:         ],  
8452:         "fiscal_year": 2025  
8453:     }  
8454:  
8455: def enhance_responsible(responsible: dict[str, Any]) -> dict[str, Any]:  
8456:     """  
8457:         Feature 7: Map Authority for Accountability  
8458:         """  
8459:     enhanced = copy.deepcopy(responsible)  
8460:  
8461:     # Add legal mandate  
8462:     entity = responsible.get('entity', '')  
8463:     if 'Mujer' in entity:  
8464:         legal_mandate = "Ley 1257 de 2008 - Normas para la prevenciÃ³n de violencias contra la mujer"
```

```
8465:     elif 'PlaneaciÃ³n' in entity:
8466:         legal_mandate = "Ley 152 de 1994 - Ley OrgÃ¡nica del Plan de Desarrollo"
8467:     elif 'Hacienda' in entity:
8468:         legal_mandate = "Ley 819 de 2003 - Responsabilidad Fiscal"
8469:     else:
8470:         legal_mandate = "Estatuto OrgÃ¡nico Municipal"
8471:
8472:     enhanced['legal_mandate'] = legal_mandate
8473:
8474:     # Add approval chain
8475:     enhanced['approval_chain'] = [
8476:         {
8477:             "level": 1,
8478:             "role": "Director/Coordinador de Programa",
8479:             "decision": "Aprueba plan de trabajo"
8480:         },
8481:         {
8482:             "level": 2,
8483:             "role": "Secretario/a de la entidad responsable",
8484:             "decision": "Aprueba presupuesto y recursos"
8485:         },
8486:         {
8487:             "level": 3,
8488:             "role": "SecretarÃ¡a de PlaneaciÃ³n",
8489:             "decision": "Valida coherencia con PDM"
8490:         },
8491:         {
8492:             "level": 4,
8493:             "role": "Alcalde Municipal",
8494:             "decision": "AprobaciÃ³n final (si aplica)"
8495:         }
8496:     ]
8497:
8498:     # Add escalation path
8499:     enhanced['escalation_path'] = {
8500:         "threshold_days_delay": 15,
8501:         "escalate_to": "SecretarÃ¡a de PlaneaciÃ³n",
8502:         "finalEscalation": "Despacho del Alcalde",
8503:         "consequences": ["RevisiÃ³n presupuestal", "ReasignaciÃ³n de responsables"]
8504:     }
8505:
8506:     return enhanced
8507:
8508: def enhance_rule(rule: dict[str, Any]) -> dict[str, Any]:
8509:     """Enhance a single rule with all 7 features"""
8510:     enhanced_rule = copy.deepcopy(rule)
8511:
8512:     # 1. Template parameterization
8513:     enhanced_rule['template'] = enhance_template(rule)
8514:
8515:     # 2. Execution logic
8516:     enhanced_rule['execution'] = add_execution_logic(rule)
8517:
8518:     # 3. Measurable indicators
8519:     if 'indicator' in enhanced_rule['template']:
8520:         enhanced_rule['template']['indicator'] = enhance_indicator(
```

```

8521:         enhanced_rule['template']['indicator'],
8522:         rule.get('rule_id', ''),
8523:         rule.get('level', '')
8524:     )
8525:
8526: # 4. Unambiguous time horizons
8527: if 'horizon' in enhanced_rule['template']:
8528:     enhanced_rule['template']['horizon'] = enhance_horizon(
8529:         enhanced_rule['template']['horizon'],
8530:         rule.get('rule_id', '')
8531:     )
8532:
8533: # 5. Testable verification
8534: if 'verification' in enhanced_rule['template']:
8535:     enhanced_rule['template']['verification'] = enhance_verification(
8536:         enhanced_rule['template']['verification'],
8537:         rule.get('rule_id', '')
8538:     )
8539:
8540: # 6. Budget tracking
8541: enhanced_rule['budget'] = add_budget(rule)
8542:
8543: # 7. Authority mapping
8544: if 'responsible' in enhanced_rule['template']:
8545:     enhanced_rule['template']['responsible'] = enhance_responsible(
8546:         enhanced_rule['template']['responsible']
8547:     )
8548:
8549: return enhanced_rule
8550:
8551: def main() -> None:
8552:     """Main enhancement process"""
8553:     # Delegate to factory for I/O operations
8554:     from farfan_pipeline.analysis.factory import load_json, save_json
8555:
8556:     # Load existing rules
8557:     rules_path = Path('config/recommendation_rules.json')
8558:     rules_data = load_json(rules_path)
8559:
8560:     print(f"Loaded {len(rules_data['rules'])} rules from {rules_path}")
8561:
8562:     # Enhance all rules
8563:     enhanced_rules = []
8564:     for i, rule in enumerate(rules_data['rules'], 1):
8565:         try:
8566:             enhanced = enhance_rule(rule)
8567:             enhanced_rules.append(enhanced)
8568:             if i % 10 == 0:
8569:                 print(f"Enhanced {i}/{len(rules_data['rules'])} rules...")
8570:         except Exception as e:
8571:             print(f"Error enhancing rule {rule.get('rule_id', 'UNKNOWN')}: {e}")
8572:             enhanced_rules.append(rule) # Keep original if enhancement fails
8573:
8574:     # Create enhanced data structure
8575:     enhanced_data = {
8576:         'version': '2.0', # Increment version

```

```
8577:     'enhanced_features': [
8578:         'template_parameterization',
8579:         'execution_logic',
8580:         'measurable_indicators',
8581:         'unambiguous_time_horizons',
8582:         'testable_verification',
8583:         'cost_tracking',
8584:         'authority_mapping'
8585:     ],
8586:     'rules': enhanced_rules
8587: }
8588:
8589: # Save enhanced rules
8590: output_path = Path('config/recommendation_rules_enhanced.json')
8591: save_json(enhanced_data, output_path)
8592:
8593: print(f"\nEnhanced {len(enhanced_rules)} rules saved to {output_path}")
8594: print(f"Original file preserved at {rules_path}")
8595:
8596: # Show sample enhanced rule
8597: if enhanced_rules:
8598:     print("\n==== Sample Enhanced Rule ====")
8599:     print(json.dumps(enhanced_rules[0], indent=2, ensure_ascii=False)[:2000])
8600:     print("...")
8601:
8602: # Note: Main entry point removed to maintain I/O boundary separation.
8603: # For usage examples, see examples/ directory.
8604:
8605:
8606:
8607: =====
8608: FILE: src/farfan_pipeline/analysis/factory.py
8609: =====
8610:
8611: """
8612: Factory module \200\224 canonical Dependency Injection (DI) and access control for F.A.R.F.A.N.
8613:
8614: This module is the single authoritative boundary for:
8615: - Canonical monolith access (CanonicalQuestionnaire)
8616: - Signal registry construction and enrichment (QuestionnaireSignalRegistry v2.0)
8617: - Method injection via MethodExecutor (with MethodRegistry + special instantiation rules)
8618: - Orchestrator construction with strict DI
8619: - CoreModuleFactory for I/O helpers (contracts, validation)
8620:
8621: Design Principles (Factory Pattern + DI):
8622: - Orchestrator and Executors never touch I/O nor load the monolith directly.
8623: - Factory loads and validates the canonical questionnaire exactly once (singleton).
8624: - Factory constructs signal registries and enriched packs centrally.
8625: - Factory wires MethodExecutor with registries and special instantiation rules.
8626: - Factory injects EnrichedSignalPack per policy area for BaseExecutor use.
8627:
8628: Scope:
8629: - Infrastructure layer only. No business logic.
8630:
8631: SIN_CARRETA Compliance:
8632: - All construction paths emit structured telemetry with timestamps and hashes.
```

```
8633: - Determinism enforced via explicit validation of canonical questionnaire integrity.
8634: - Contract assertions guard all factory outputs (no silent degradation).
8635: - Auditability via immutable ProcessorBundle with provenance metadata.
8636: - SeedRegistry singleton ensures deterministic stochastic operations.
8637:
8638: Integration Points:
8639: 1. Orchestrator receives: method_executor, questionnaire, executor_config
8640: 2. BaseExecutor (30 classes) receives: enriched_signal_pack (via helper function)
8641: 3. MethodExecutor routes all method calls via ExtendedArgRouter
8642: 4. Special instantiation rules enable shared MunicipalOntology, dependency injection
8643: """
8644:
8645: from __future__ import annotations
8646:
8647: import hashlib
8648: import json
8649: import logging
8650: import time
8651: from dataclasses import dataclass, field
8652: from pathlib import Path
8653: from typing import Any, Optional
8654:
8655: from farfan_pipeline.core.orchestrator.core import MethodExecutor
8656: from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
8657: from farfan_pipeline.core.orchestrator.method_registry import (
8658:     MethodRegistry,
8659:     setup_default_instantiation_rules,
8660: )
8661: from farfan_pipeline.core.orchestrator.signal_registry import (
8662:     QuestionnaireSignalRegistry,
8663:     create_signal_registry,
8664: )
8665: from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
8666:     EnrichedSignalPack,
8667:     create_enriched_signal_pack,
8668: )
8669: from farfan_pipeline.core.orchestrator.questionnaire import (
8670:     CanonicalQuestionnaire,
8671:     load_questionnaire,
8672: )
8673: from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
8674: from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
8675:
8676: # Optional: CoreModuleFactory for I/O helpers
8677: try:
8678:     from farfan_pipeline.core.orchestrator.core_module_factory import CoreModuleFactory
8679:     CORE_MODULE_FACTORY_AVAILABLE = True
8680: except ImportError:
8681:     CoreModuleFactory = None # type: ignore
8682:     CORE_MODULE_FACTORY_AVAILABLE = False
8683:
8684: # Optional: SeedRegistry for determinism
8685: try:
8686:     from farfan_pipeline.core.orchestrator.seed_registry import SeedRegistry
8687:     SEED_REGISTRY_AVAILABLE = True
8688: except ImportError:
```

```
8689:     SeedRegistry = None # type: ignore
8690:     SEED_REGISTRY_AVAILABLE = False
8691:
8692:     logger = logging.getLogger(__name__)
8693:
8694:
8695:     # =====
8696:     # Exceptions
8697:     # =====
8698:
8699:
8700: class FactoryError(Exception):
8701:     """Base exception for factory construction failures."""
8702:     pass
8703:
8704:
8705: class QuestionnaireValidationError(FactoryError):
8706:     """Raised when questionnaire validation fails."""
8707:     pass
8708:
8709:
8710: class RegistryConstructionError(FactoryError):
8711:     """Raised when signal registry construction fails."""
8712:     pass
8713:
8714:
8715: class ExecutorConstructionError(FactoryError):
8716:     """Raised when method executor construction fails."""
8717:     pass
8718:
8719:
8720: # =====
8721: # Processor Bundle (typed DI container with provenance)
8722: # =====
8723:
8724:
8725: @dataclass(frozen=True)
8726: class ProcessorBundle:
8727:     """Aggregated orchestrator dependencies built by the Factory.
8728:
8729:     Attributes:
8730:         method_executor: Preconfigured MethodExecutor ready for routing.
8731:         questionnaire: Immutable, validated CanonicalQuestionnaire.
8732:         signal_registry: QuestionnaireSignalRegistry v2.0 with full metadata.
8733:         executor_config: Canonical ExecutorConfig for executors.
8734:         enriched_signal_packs: Dict of EnrichedSignalPack per policy area.
8735:         core_module_factory: Optional CoreModuleFactory for I/O helpers.
8736:         provenance: Construction metadata for audit trails.
8737:     """
8738:
8739:     method_executor: MethodExecutor
8740:     questionnaire: CanonicalQuestionnaire
8741:     signal_registry: QuestionnaireSignalRegistry
8742:     executor_config: ExecutorConfig
8743:     enriched_signal_packs: dict[str, EnrichedSignalPack]
8744:     core_module_factory: Optional[Any] = None
```

```
8745:     provenance: dict[str, Any] = field(default_factory=dict)
8746:
8747:     def __post_init__(self) -> None:
8748:         """SIN_CARRETA Â$ Contract Enforcement: validate bundle integrity."""
8749:         errors = []
8750:
8751:         if self.method_executor is None:
8752:             errors.append("method_executor must not be None")
8753:         if self.questionnaire is None:
8754:             errors.append("questionnaire must not be None")
8755:         if self.signal_registry is None:
8756:             errors.append("signal_registry must not be None")
8757:         if self.executor_config is None:
8758:             errors.append("executor_config must not be None")
8759:         if self.enriched_signal_packs is None:
8760:             errors.append("enriched_signal_packs must not be None")
8761:         elif not isinstance(self.enriched_signal_packs, dict):
8762:             errors.append("enriched_signal_packs must be dict[str, EnrichedSignalPack]")
8763:
8764:         if not self.provenance.get("construction_timestamp_utc"):
8765:             errors.append("provenance must include construction_timestamp_utc")
8766:         if not self.provenance.get("canonical_sha256"):
8767:             errors.append("provenance must include canonical_sha256")
8768:         if self.provenance.get("signal_registry_version") != "2.0":
8769:             errors.append("provenance must indicate signal_registry_version=2.0")
8770:
8771:         if errors:
8772:             raise FactoryError(f"ProcessorBundle validation failed: {'; '.join(errors)}")
8773:
8774:         logger.info(
8775:             "processor_bundle_validated "
8776:             "canonical_sha256=%s construction_ts=%s policy_areas=%d",
8777:             self.provenance.get("canonical_sha256", "")[:16],
8778:             self.provenance.get("construction_timestamp_utc"),
8779:             len(self.enriched_signal_packs),
8780:         )
8781:
8782:
8783: # =====
8784: # Core Factory Implementation
8785: # =====
8786:
8787:
8788: def build_processor_bundle(
8789:     *,
8790:     questionnaire_path: Optional[str] = None,
8791:     executor_config: Optional[ExecutorConfig] = None,
8792:     enable_intelligence_layer: bool = True,
8793:     seed_for_determinism: Optional[int] = None,
8794:     strict_validation: bool = True,
8795: ) -> ProcessorBundle:
8796:     """Build complete processor bundle with all dependencies wired.
8797:
8798:     This is the primary factory entry point for constructing all orchestrator
8799:     dependencies in a single, validated operation.
8800:
```

```
8801:     Args:
8802:         questionnaire_path: Path to canonical questionnaire JSON. If None, uses default.
8803:         executor_config: Custom executor configuration. If None, uses default.
8804:         enable_intelligence_layer: Whether to build enriched signal packs (default: True).
8805:         seed_for_determinism: Optional seed for reproducible stochastic operations.
8806:         strict_validation: If True, fail on any validation error (default: True).
8807:
8808:     Returns:
8809:         ProcessorBundle: Immutable bundle with all dependencies wired and validated.
8810:
8811:     Raises:
8812:         QuestionnaireValidationException: If questionnaire validation fails.
8813:         RegistryConstructionException: If signal registry construction fails.
8814:         ExecutorConstructionException: If method executor construction fails.
8815:         FactoryException: For other construction failures.
8816:     """
8817:     construction_start = time.time()
8818:     timestamp_utc = time.strftime("%Y-%m-%dT%H:%M:%S", time.gmtime())
8819:
8820:     logger.info("factory_build_start timestamp=%s strict=%s", timestamp_utc, strict_validation)
8821:
8822:     try:
8823:         # Step 1: Load and validate canonical questionnaire
8824:         questionnaire = _load_and_validate_questionnaire(questionnaire_path, strict_validation)
8825:         canonical_hash = _compute_questionnaire_hash(questionnaire)
8826:
8827:         if not isinstance(questionnaire, CanonicalQuestionnaire):
8828:             logger.error("Loaded questionnaire is not a CanonicalQuestionnaire instance: type=%s", type(questionnaire))
8829:             num_questions = 0
8830:         elif not hasattr(questionnaire, 'questions') or not isinstance(questionnaire.questions, (list, tuple)):
8831:             logger.error("CanonicalQuestionnaire missing 'questions' attribute or it is not a list/tuple: %s", repr(questionnaire))
8832:             num_questions = 0
8833:         else:
8834:             num_questions = len(questionnaire.questions)
8835:             logger.info(
8836:                 "questionnaire_loaded questions=%d hash=%s",
8837:                 num_questions,
8838:                 canonical_hash[:16],
8839:             )
8840:
8841:         # Step 2: Build signal registry v2.0
8842:         signal_registry = _build_signal_registry(questionnaire, strict_validation)
8843:
8844:         if not hasattr(signal_registry, 'get_all_policy_areas') or not callable(getattr(signal_registry, 'get_all_policy_areas', None)):
8845:             logger.error("signal_registry does not implement required method 'get_all_policy_areas'")
8846:             raise AttributeError("signal_registry does not implement required method 'get_all_policy_areas'")
8847:         logger.info(
8848:             "signal_registry_built version=2.0 policy_areas=%d",
8849:             len(signal_registry.get_all_policy_areas()),
8850:         )
8851:
8852:         # Step 3: Build enriched signal packs (intelligence layer)
8853:         enriched_packs = _build_enriched_packs(
8854:             signal_registry,
8855:             questionnaire,
8856:             enable_intelligence_layer,
```

```
8857:         strict_validation
8858:     )
8859:
8860:     logger.info(
8861:         "enriched_packs_built count=%d intelligence_layer=%s",
8862:         len(enriched_packs),
8863:         "enabled" if enable_intelligence_layer else "disabled",
8864:     )
8865:
8866:     # Step 4: Initialize seed registry for determinism
8867:     _initialize_seed_registry(seed_for_determinism)
8868:
8869:     # Step 5: Build method executor with full wiring
8870:     method_executor = _build_method_executor(strict_validation)
8871:
8872:     logger.info(
8873:         "method_executor_built special_routes=%d",
8874:         method_executor.arg_router.get_special_route_coverage() if hasattr(method_executor.arg_router, 'get_special_route_coverage') else 0,
8875:     )
8876:
8877:     # Step 6: Build or use provided executor config
8878:     if executor_config is None:
8879:         executor_config = ExecutorConfig.default()
8880:
8881:     # Step 7: Build optional core module factory
8882:     core_factory = _build_core_module_factory()
8883:
8884:     # Step 8: Assemble provenance metadata
8885:     construction_duration = time.time() - construction_start
8886:     provenance = {
8887:         "construction_timestamp_utc": timestamp_utc,
8888:         "canonical_sha256": canonical_hash,
8889:         "signal_registry_version": "2.0",
8890:         "intelligence_layer_enabled": enable_intelligence_layer,
8891:         "enriched_packs_count": len(enriched_packs),
8892:         "construction_duration_seconds": round(construction_duration, 3),
8893:         "seed_registry_initialized": SEED_REGISTRY_AVAILABLE and seed_for_determinism is not None,
8894:         "core_module_factory_available": CORE_MODULE_FACTORY_AVAILABLE,
8895:         "strict_validation": strict_validation,
8896:     }
8897:
8898:     # Step 9: Build and validate bundle
8899:     bundle = ProcessorBundle(
8900:         method_executor=method_executor,
8901:         questionnaire=questionnaire,
8902:         signal_registry=signal_registry,
8903:         executor_config=executor_config,
8904:         enriched_signal_packs=enriched_packs,
8905:         core_module_factory=core_factory,
8906:         provenance=provenance,
8907:     )
8908:
8909:     logger.info(
8910:         "factory_build_complete duration=%.3fs hash=%s",
8911:         construction_duration,
8912:         canonical_hash[:16],
```

```
8913:         )
8914:
8915:     return bundle
8916:
8917: except Exception as e:
8918:     logger.error("factory_build_failed error=%s", str(e), exc_info=True)
8919:     raise FactoryError(f"Failed to build processor bundle: {e}") from e
8920:
8921:
8922: # =====
8923: # Internal Construction Functions
8924: # =====
8925:
8926:
8927: def _load_and_validate_questionnaire(
8928:     path: Optional[str],
8929:     strict: bool,
8930: ) -> CanonicalQuestionnaire:
8931:     """Load and validate canonical questionnaire."""
8932:     try:
8933:         questionnaire_path = Path(path) if path is not None else None
8934:         questionnaire = load_questionnaire(questionnaire_path)
8935:
8936:         # Validate structure
8937:         if not hasattr(questionnaire, 'questions'):
8938:             if strict:
8939:                 raise QuestionnaireValidationError("Questionnaire missing 'questions' attribute")
8940:             logger.warning("questionnaire_validation_warning_missing_questions_attribute")
8941:
8942:         questions = getattr(questionnaire, 'questions', [])
8943:         if not questions:
8944:             if strict:
8945:                 raise QuestionnaireValidationError("Questionnaire has no questions")
8946:             logger.warning("questionnaire_validation_warning_no_questions")
8947:
8948:     return questionnaire
8949:
8950: except Exception as e:
8951:     if strict:
8952:         raise QuestionnaireValidationError(f"Failed to load questionnaire: {e}") from e
8953:     logger.error("questionnaire_load_error_continuing_with_degraded_state", exc_info=True)
8954:     raise
8955:
8956:
8957: def _build_signal_registry(
8958:     questionnaire: CanonicalQuestionnaire,
8959:     strict: bool,
8960: ) -> QuestionnaireSignalRegistry:
8961:     """Build signal registry from questionnaire."""
8962:     try:
8963:         registry = create_signal_registry(questionnaire)
8964:
8965:         # Validate registry
8966:         if not hasattr(registry, 'get_all_policy_areas'):
8967:             if strict:
8968:                 raise RegistryConstructionError("Registry missing required methods")
```

```
8969:         logger.warning("registry_validation_warning missing_methods")
8970:
8971:     return registry
8972:
8973: except Exception as e:
8974:     if strict:
8975:         raise RegistryConstructionError(f"Failed to build signal registry: {e}") from e
8976:     logger.error("registry_construction_error", exc_info=True)
8977:     raise
8978:
8979:
8980: def _build_enriched_packs(
8981:     signal_registry: QuestionnaireSignalRegistry,
8982:     questionnaire: CanonicalQuestionnaire,
8983:     enable: bool,
8984:     strict: bool,
8985: ) -> dict[str, EnrichedSignalPack]:
8986:     """Build enriched signal packs for all policy areas."""
8987:     enriched_packs: dict[str, EnrichedSignalPack] = {}
8988:
8989:     if not enable:
8990:         logger.info("enriched_packs_disabled")
8991:     return enriched_packs
8992:
8993:     try:
8994:         policy_areas = signal_registry.get_all_policy_areas() if hasattr(signal_registry, 'get_all_policy_areas') else []
8995:
8996:         if not policy_areas:
8997:             logger.warning("no_policy_areas_found registry_empty")
8998:         return enriched_packs
8999:
9000:     for policy_area_id in policy_areas:
9001:         try:
9002:             base_pack = signal_registry.get(policy_area_id) if hasattr(signal_registry, 'get') else None
9003:
9004:             if base_pack is None:
9005:                 logger.warning("base_pack_missing policy_area=%s", policy_area_id)
9006:                 continue
9007:
9008:             enriched_pack = create_enriched_signal_pack(
9009:                 base_pack,
9010:                 questionnaire,
9011:             )
9012:             enriched_packs[policy_area_id] = enriched_pack
9013:
9014:         except Exception as e:
9015:             msg = f"Failed to create enriched pack for {policy_area_id}: {e}"
9016:             if strict:
9017:                 raise RegistryConstructionError(msg) from e
9018:             logger.error("enriched_pack_creation_failed policy_area=%s", policy_area_id, exc_info=True)
9019:
9020:     return enriched_packs
9021:
9022: except Exception as e:
9023:     if strict:
9024:         raise RegistryConstructionError(f"Failed to build enriched packs: {e}") from e
```

```
9025:         logger.error("enriched_packs_construction_error", exc_info=True)
9026:         return enriched_packs
9027:
9028:
9029: def _initialize_seed_registry(seed: Optional[int]) -> None:
9030:     """Initialize seed registry if available."""
9031:     if not SEED_REGISTRY_AVAILABLE:
9032:         logger.debug("seed_registry_unavailable module_not_found")
9033:         return
9034:
9035:     if seed is None:
9036:         logger.debug("seed_registry_not_initialized no_seed_provided")
9037:         return
9038:
9039:     try:
9040:         SeedRegistry.initialize(master_seed=seed)
9041:         logger.info("seed_registry_initialized master_seed=%d", seed)
9042:     except Exception as e:
9043:         logger.error("seed_registry_initialization_failed", exc_info=True)
9044:         # Non-fatal, continue without determinism
9045:
9046:
9047: def _build_method_executor(strict: bool) -> MethodExecutor:
9048:     """Build method executor with full dependency wiring."""
9049:     try:
9050:         # Build method registry
9051:         method_registry = MethodRegistry()
9052:         setup_default_instantiation_rules(method_registry)
9053:
9054:         # Build class registry
9055:         class_registry = build_class_registry()
9056:
9057:         # Build extended arg router
9058:         arg_router = ExtendedArgRouter(class_registry)
9059:
9060:         # Build method executor
9061:         method_executor = MethodExecutor(
9062:             method_registry=method_registry,
9063:         )
9064:
9065:         # Validate construction
9066:         if not hasattr(method_executor, 'execute'):
9067:             if strict:
9068:                 raise ExecutorConstructionError("MethodExecutor missing 'execute' method")
9069:             logger.warning("method_executor_validation_warning missing_execute")
9070:
9071:         return method_executor
9072:
9073:     except Exception as e:
9074:         if strict:
9075:             raise ExecutorConstructionError(f"Failed to build method executor: {e}") from e
9076:         logger.error("method_executor_construction_error", exc_info=True)
9077:         raise
9078:
9079:
9080: def _build_core_module_factory() -> Optional[Any]:
```

```
9081:     """Build core module factory if available."""
9082:     if not CORE_MODULE_FACTORY_AVAILABLE:
9083:         logger.debug("core_module_factory_unavailable module_not_found")
9084:         return None
9085:
9086:     try:
9087:         factory = CoreModuleFactory()
9088:         logger.info("core_module_factory_built")
9089:         return factory
9090:     except Exception as e:
9091:         logger.error("core_module_factory_construction_error", exc_info=True)
9092:         return None
9093:
9094:
9095: def _compute_questionnaire_hash(questionnaire: CanonicalQuestionnaire) -> str:
9096:     """Compute deterministic SHA256 hash of questionnaire content."""
9097:     try:
9098:         # Try to get JSON representation if available
9099:         if hasattr(questionnaire, 'to_dict'):
9100:             content = json.dumps(questionnaire.to_dict(), sort_keys=True)
9101:         elif hasattr(questionnaire, '__dict__'):
9102:             content = json.dumps(questionnaire.__dict__, sort_keys=True, default=str)
9103:         else:
9104:             content = str(questionnaire)
9105:
9106:         return hashlib.sha256(content.encode('utf-8')).hexdigest()
9107:
9108:     except Exception as e:
9109:         logger.warning("questionnaire_hash_computation_degraded error=%s", str(e))
9110:         # Fallback to simple string hash
9111:         return hashlib.sha256(str(questionnaire).encode('utf-8')).hexdigest()
9112:
9113:
9114: # =====
9115: # Convenience API
9116: # =====
9117:
9118:
9119: def build_processor(
9120:     questionnaire_path: Optional[str] = None,
9121:     seed: Optional[int] = None,
9122: ) -> ProcessorBundle:
9123:     """
9124:     Convenience wrapper for 'build_processor_bundle' with sensible defaults.
9125:
9126:     This function is intended for typical use cases where you want a fully configured
9127:     processor with the intelligence layer enabled, strict validation, and optional
9128:     reproducibility via a seed. It sets recommended defaults for most users.
9129:
9130:     Use 'build_processor_bundle' directly if you need advanced customization, such as
9131:     disabling the intelligence layer, changing validation strictness, or other options.
9132:
9133:     Args:
9134:         questionnaire_path: Optional path to questionnaire JSON.
9135:         seed: Optional seed for reproducibility.
9136:     Returns:
```

```
9137:         ProcessorBundle ready for use.
9138:     """
9139:     return build_processor_bundle(
9140:         questionnaire_path=questionnaire_path,
9141:         enable_intelligence_layer=True,
9142:         seed_for_determinism=seed,
9143:         strict_validation=True,
9144:     )
9145:
9146:
9147: def build_minimal_processor(
9148:     questionnaire_path: Optional[str] = None,
9149:     strict: bool = False,
9150: ) -> ProcessorBundle:
9151:     """Build minimal processor bundle without intelligence layer.
9152:
9153:     Useful for testing or when enriched signals are not needed.
9154:
9155:     Args:
9156:         questionnaire_path: Optional path to questionnaire JSON.
9157:         strict: Whether to use strict validation (default: False for minimal).
9158:
9159:     Returns:
9160:         ProcessorBundle with basic dependencies only.
9161:     """
9162:     return build_processor_bundle(
9163:         questionnaire_path=questionnaire_path,
9164:         enable_intelligence_layer=False,
9165:         strict_validation=strict,
9166:     )
9167:
9168:
9169: def get_enriched_pack_for_policy_area(
9170:     bundle: ProcessorBundle,
9171:     policy_area_id: str,
9172: ) -> Optional[EnrichedSignalPack]:
9173:     """Helper to safely retrieve enriched signal pack from bundle.
9174:
9175:     Args:
9176:         bundle: Processor bundle.
9177:         policy_area_id: Policy area identifier.
9178:
9179:     Returns:
9180:         EnrichedSignalPack if available, None otherwise.
9181:     """
9182:     return bundle.enriched_signal_packs.get(policy_area_id)
9183:
9184:
9185: # =====
9186: # Validation and Diagnostics
9187: # =====
9188:
9189:
9190: def validate_bundle(bundle: ProcessorBundle) -> dict[str, Any]:
9191:     """Validate bundle integrity and return diagnostics.
9192:
```

```
9193:     Args:
9194:         bundle: ProcessorBundle to validate.
9195:
9196:     Returns:
9197:         Dictionary with validation results and diagnostics.
9198:     """
9199:     diagnostics = {
9200:         "valid": True,
9201:         "errors": [],
9202:         "warnings": [],
9203:         "components": {},
9204:         "metrics": {},
9205:     }
9206:
9207:     # Validate method executor
9208:     if bundle.method_executor is None:
9209:         diagnostics["valid"] = False
9210:         diagnostics["errors"].append("method_executor is None")
9211:     else:
9212:         diagnostics["components"]["method_executor"] = "present"
9213:         if hasattr(bundle.method_executor, 'arg_router'):
9214:             router = bundle.method_executor.arg_router
9215:             if hasattr(router, 'get_special_route_coverage'):
9216:                 diagnostics["metrics"]["special_routes"] = router.get_special_route_coverage()
9217:
9218:     # Validate questionnaire
9219:     if bundle.questionnaire is None:
9220:         diagnostics["valid"] = False
9221:         diagnostics["errors"].append("questionnaire is None")
9222:     else:
9223:         diagnostics["components"]["questionnaire"] = "present"
9224:         if hasattr(bundle.questionnaire, 'questions'):
9225:             diagnostics["metrics"]["question_count"] = len(bundle.questionnaire.questions)
9226:
9227:     # Validate signal registry
9228:     if bundle.signal_registry is None:
9229:         diagnostics["valid"] = False
9230:         diagnostics["errors"].append("signal_registry is None")
9231:     else:
9232:         diagnostics["components"]["signal_registry"] = "present"
9233:         if hasattr(bundle.signal_registry, 'get_all_policy_areas'):
9234:             diagnostics["metrics"]["policy_areas"] = len(bundle.signal_registry.get_all_policy_areas())
9235:
9236:     # Validate enriched packs
9237:     pack_count = len(bundle.enriched_signal_packs)
9238:     diagnostics["components"]["enriched_packs"] = pack_count
9239:     diagnostics["metrics"]["enriched_pack_count"] = pack_count
9240:
9241:     if pack_count == 0 and bundle.provenance.get("intelligence_layer_enabled"):
9242:         diagnostics["warnings"].append("Intelligence layer enabled but no enriched packs available")
9243:
9244:     # Validate provenance
9245:     required_provenance = ["construction_timestamp_utc", "canonical_sha256", "signal_registry_version"]
9246:     missing_provenance = [k for k in required_provenance if k not in bundle.provenance]
9247:     if missing_provenance:
9248:         diagnostics["valid"] = False
```

```
9249:         diagnostics["errors"].append(f"Missing provenance: {missing_provenance}")
9250:
9251:     # Check provenance metrics
9252:     diagnostics["metrics"]["construction_duration"] = bundle.provenance.get("construction_duration_seconds", 0)
9253:     diagnostics["metrics"]["canonical_hash"] = bundle.provenance.get("canonical_sha256", "")[:16]
9254:
9255:     return diagnostics
9256:
9257:
9258: def get_bundle_info(bundle: ProcessorBundle) -> dict[str, Any]:
9259:     """Get human-readable information about bundle.
9260:
9261:     Args:
9262:         bundle: ProcessorBundle to inspect.
9263:
9264:     Returns:
9265:         Dictionary with bundle information.
9266:     """
9267:     return {
9268:         "construction_time": bundle.provenance.get("construction_timestamp_utc"),
9269:         "canonical_hash": bundle.provenance.get("canonical_sha256", "")[:16],
9270:         "policy_areas": sorted(bundle.enriched_signal_packs.keys()),
9271:         "policy_area_count": len(bundle.enriched_signal_packs),
9272:         "intelligence_layer": bundle.provenance.get("intelligence_layer_enabled"),
9273:         "core_factory": bundle.core_module_factory is not None,
9274:         "construction_duration": bundle.provenance.get("construction_duration_seconds"),
9275:         "strict_validation": bundle.provenance.get("strict_validation"),
9276:         "seed_initialized": bundle.provenance.get("seed_registry_initialized"),
9277:     }
9278:
9279:
9280: # =====
9281: # Singleton Cache (Optional)
9282: # =====
9283:
9284: _bundle_cache: Optional[ProcessorBundle] = None
9285: _cache_key: Optional[str] = None
9286:
9287:
9288: def get_or_build_bundle(
9289:     questionnaire_path: Optional[str] = None,
9290:     cache: bool = True,
9291:     force_rebuild: bool = False,
9292: ) -> ProcessorBundle:
9293:     """Get cached bundle or build new one.
9294:
9295:     Args:
9296:         questionnaire_path: Optional path to questionnaire JSON.
9297:         cache: Whether to cache the bundle (default: True).
9298:         force_rebuild: Force rebuild even if cached (default: False).
9299:
9300:     Returns:
9301:         ProcessorBundle (cached or newly built).
9302:     """
9303:     global _bundle_cache, _cache_key
9304:
```

```
9305:     cache_key = questionnaire_path or "default"
9306:
9307:     if not force_rebuild and cache and _bundle_cache is not None and _cache_key == cache_key:
9308:         logger.debug("factory_cache_hit key=%s", cache_key)
9309:         return _bundle_cache
9310:
9311:     logger.debug("factory_cache_miss key=%s building_new force_rebuild=%s", cache_key, force_rebuild)
9312:     bundle = build_processor(questionnaire_path=questionnaire_path)
9313:
9314:     if cache:
9315:         _bundle_cache = bundle
9316:         _cache_key = cache_key
9317:         logger.debug("factory_cache_updated key=%s", cache_key)
9318:
9319:     return bundle
9320:
9321:
9322: def clear_bundle_cache() -> None:
9323:     """Clear singleton bundle cache."""
9324:     global _bundle_cache, _cache_key
9325:     _bundle_cache = None
9326:     _cache_key = None
9327:     logger.debug("factory_cache_cleared")
9328:
9329:
9330: def get_cache_info() -> dict[str, Any]:
9331:     """Get information about current cache state."""
9332:     return {
9333:         "cached": _bundle_cache is not None,
9334:         "cache_key": _cache_key,
9335:         "bundle_hash": _bundle_cache.provenance.get("canonical_sha256", "")[:16] if _bundle_cache else None,
9336:     }
9337:
9338:
9339:
9340: =====
9341: FILE: src/farfan_pipeline/analysis/financiero_viability_tablas.py
9342: =====
9343:
9344: """
9345: MUNICIPAL DEVELOPMENT PLAN ANALYZER - PDET COLOMBIA
9346: =====
9347: VersiÃ³n: 5.0 - Causal Inference Edition (2025)
9348: EspecializaciÃ³n: Planes de Desarrollo Municipal con AnÃ¡lisis Causal Bayesiano
9349: Arquitectura: ExtracciÃ³n Avanzada + Inferencia Causal + DAG Learning + Counterfactuals
9350:
9351: NUEVA CAPACIDAD - INFERENCIA CAUSAL:
9352: \234\223 IdentificaciÃ³n automÃ;tica de mecanismos causales en PDM
9353: \234\223 ConstrucciÃ³n de DAGs (Directed Acyclic Graphs) para pilares PDET
9354: \234\223 EstimaciÃ³n bayesiana de efectos causales directos e indirectos
9355: \234\223 AnÃ¡lisis contrafactual de intervenciones
9356: \234\223 CuantificaciÃ³n de heterogeneidad causal por contexto territorial
9357: \234\223 DetecciÃ³n de confounders y mediadores
9358: \234\223 AnÃ¡lisis de sensibilidad para supuestos de identificaciÃ³n
9359:
9360: COMPLIANCE:
```

```
9361: #\234\223 Python 3.10+ con type hints completos
9362: #\234\223 Sin placeholders - 100% implementado y probado
9363: #\234\223 IntegraciÃ³n completa con pipeline existente
9364: #\234\223 Calibrado para estructura de PDM colombianos
9365: """
9366: from __future__ import annotations
9367:
9368: import asyncio
9369: import logging
9370: import re
9371: from dataclasses import dataclass, field
9372: from datetime import datetime
9373: from decimal import Decimal
9374: from pathlib import Path
9375: from typing import Any, Literal
9376:
9377: # === EXTRACCIÃ\223N AVANZADA DE PDF Y TABLAS ===
9378: import camelot
9379:
9380: # === NETWORKING Y GRAFOS CAUSALES ===
9381: import networkx as nx
9382:
9383: # === CORE SCIENTIFIC COMPUTING ===
9384: import numpy as np
9385: import pandas as pd
9386:
9387: # === ESTADÃ\215STICA BAYESIANA Y CAUSAL INFERENCE ===
9388: import pymc as pm
9389: import spacy
9390: import tabula
9391: import torch
9392: from scipy import stats
9393:
9394: # === NLP Y TRANSFORMERS ===
9395: # Check dependency lockdown before importing transformers
9396: from farfan_pipeline.core.dependency_lockdown import get_dependency_lockdown
9397: from sentence_transformers import SentenceTransformer, util
9398: from sklearn.cluster import DBSCAN, AgglomerativeClustering
9399:
9400: # === MACHINE LEARNING Y SCORING ===
9401: from sklearn.feature_extraction.text import TfidfVectorizer
9402: from transformers import pipeline
9403: from farfan_pipeline.core.parameters import ParameterLoaderV2
9404: from farfan_pipeline.core.calibration.decorators import calibrated_method
9405:
9406: _lockdown = get_dependency_lockdown()
9407:
9408: # =====
9409: # LOGGING CONFIGURATION
9410: # =====
9411: logger = logging.getLogger(__name__)
9412:
9413: # =====
9414: # CONFIGURACIÃ\223N ESPECÃ\215FICA PARA COLOMBIA Y PDET
9415: # =====
9416:
```

```

9417: class ColombianMunicipalContext:
9418:     """Contexto específico del marco normativo colombiano para PDM"""
9419:
9420:     OFFICIAL_SYSTEMS: dict[str, str] = {
9421:         'SISBEN': r'SISB[É\211]N\s*(?:I{1,4}|IV)?',
9422:         'SGP': r'Sistema\s+General\s+de\s+Participaciones|SGP',
9423:         'SGR': r'Sistema\s+General\s+de\s+Regal[Á-i]as|SGR',
9424:         'FUT': r'Formulario\s+[Á\232U]nico\s+Territorial|FUT',
9425:         'MFMP': r'Marco\s+Fiscal\s+(?:de\s+)?Mediano\s+Plazo|MFMP',
9426:         'CONPES': r'CONPES\s*\d{3,4}',
9427:         'DANE': r'(?DANE|C[Á³o]digo\s+DANE)\s*[:\-\-]\s*(\d{5,8})',
9428:         'MGA': r'Metodolog[Á-i]a\s+General\s+Ajustada|MGA',
9429:         'POAI': r'Plan\s+Operativo\s+Anual\s+de\s+Inversiones|POAI'
9430:     }
9431:
9432:     TERRITORIAL_CATEGORIES: dict[int, dict[str, Any]] = {
9433:         1: {'name': 'Especial', 'min_pop': 500_001, 'min_income_smmlv': 400_000},
9434:         2: {'name': 'Primera', 'min_pop': 100_001, 'min_income_smmlv': 100_000},
9435:         3: {'name': 'Segunda', 'min_pop': 50_001, 'min_income_smmlv': 50_000},
9436:         4: {'name': 'Tercera', 'min_pop': 30_001, 'min_income_smmlv': 30_000},
9437:         5: {'name': 'Cuarta', 'min_pop': 20_001, 'min_income_smmlv': 25_000},
9438:         6: {'name': 'Quinta', 'min_pop': 10_001, 'min_income_smmlv': 15_000},
9439:         7: {'name': 'Sexta', 'min_pop': 0, 'min_income_smmlv': 0}
9440:     }
9441:
9442:     DNP_DIMENSIONS: list[str] = [
9443:         'DimensiÃ³n EconÃ³mica',
9444:         'DimensiÃ³n Social',
9445:         'DimensiÃ³n Ambiental',
9446:         'DimensiÃ³n Institucional',
9447:         'DimensiÃ³n Territorial'
9448:     ]
9449:
9450:     PDET_PILLARS: list[str] = [
9451:         'Ordenamiento social de la propiedad rural',
9452:         'Infraestructura y adecuaciÃ³n de tierras',
9453:         'Salud rural',
9454:         'EducaciÃ³n rural y primera infancia',
9455:         'Vivienda, agua potable y saneamiento bÃ¡sico',
9456:         'ReactivaciÃ³n econÃ³mica y producciÃ³n agropecuaria',
9457:         'Sistema para la garantÃ-a progresiva del derecho a la alimentaciÃ³n',
9458:         'ReconciliaciÃ³n, convivencia y paz'
9459:     ]
9460:
9461:     PDET THEORY_OF_CHANGE: dict[str, dict[str, Any]] = {
9462:         'Ordenamiento social de la propiedad rural': {
9463:             'outcomes': ['seguridad_juridica', 'reducción_conflictos_tienda'],
9464:             'mediators': ['formalización', 'acceso_justicia'],
9465:             'lag_years': 3
9466:         },
9467:         'Infraestructura y adecuaciÃ³n de tierras': {
9468:             'outcomes': ['conectividad', 'productividad_agricola'],
9469:             'mediators': ['vias_terciarias', 'distritos_riege'],
9470:             'lag_years': 2
9471:         },
9472:         'Salud rural': {

```

```

9473:         'outcomes': ['mortalidad_infantil', 'esperanza_vida'],
9474:         'mediators': ['cobertura_salud', 'infraestructura_salud'],
9475:         'lag_years': 4
9476:     },
9477:     'EducaciÃ³n rural y primera infancia': {
9478:         'outcomes': ['cobertura_educativa', 'calidad_educativa'],
9479:         'mediators': ['infraestructura_escolar', 'docentes_calificados'],
9480:         'lag_years': 5
9481:     },
9482:     'Vivienda, agua potable y saneamiento bÃ¡sico': {
9483:         'outcomes': ['deficit_habitacional', 'enfermedades_hidricas'],
9484:         'mediators': ['cobertura_acueducto', 'viviendas_dignas'],
9485:         'lag_years': 3
9486:     },
9487:     'ReactivaciÃ³n econÃ³mica y producciÃ³n agropecuaria': {
9488:         'outcomes': ['ingreso_rural', 'empleo_rural'],
9489:         'mediators': ['credito_rural', 'asistencia_tecnica'],
9490:         'lag_years': 2
9491:     },
9492:     'Sistema para la garantÃ-a progresiva del derecho a la alimentaciÃ³n': {
9493:         'outcomes': ['seguridad_alimentaria', 'nutricion_infantil'],
9494:         'mediators': ['produccion_local', 'acceso_alimentos'],
9495:         'lag_years': 2
9496:     },
9497:     'ReconciliaciÃ³n, convivencia y paz': {
9498:         'outcomes': ['cohesion_social', 'confianza_institucional'],
9499:         'mediators': ['espacios_participacion', 'justicia_transicional'],
9500:         'lag_years': 6
9501:     }
9502: }
9503:
9504: INDICATOR_STRUCTURE: dict[str, list[str]] = {
9505:     'resultado': ['lÃ-nea_base', 'meta', 'aÃ±o_base', 'aÃ±o_meta', 'fuente', 'responsable'],
9506:     'producto': ['indicador', 'fÃ³rmula', 'unidad_medida', 'lÃ-nea_base', 'meta', 'periodicidad'],
9507:     'gestiÃ³n': ['eficacia', 'eficiencia', 'economÃ-a', 'costo_beneficio']
9508: }
9509:
9510: # =====
9511: # ESTRUCTURAS DE DATOS
9512: # =====
9513:
9514: @dataclass
9515: class CausalNode:
9516:     """Nodo en el grafo causal"""
9517:     name: str
9518:     node_type: Literal['pilar', 'outcome', 'mediator', 'confounder']
9519:     embedding: np.ndarray | None = None
9520:     associated_budget: Decimal | None = None
9521:     temporal_lag: int = 0
9522:     evidence_strength: float = 0.0
9523:
9524: @dataclass
9525: class CausalEdge:
9526:     """Arista causal entre nodos"""
9527:     source: str
9528:     target: str

```

```
9529:     edge_type: Literal['direct', 'mediated', 'confounded']
9530:     effect_size_posterior: tuple[float, float, float] | None = None
9531:     mechanism: str = ""
9532:     evidence_quotes: list[str] = field(default_factory=list)
9533:     probability: float = 0.0
9534:
9535: @dataclass
9536: class CausalDAG:
9537:     """Grafo AcÃ¡lico Dirigido completo"""
9538:     nodes: dict[str, CausalNode]
9539:     edges: list[CausalEdge]
9540:     adjacency_matrix: np.ndarray
9541:     graph: nx.DiGraph
9542:
9543: @dataclass
9544: class CausalEffect:
9545:     """Efecto causal estimado"""
9546:     treatment: str
9547:     outcome: str
9548:     effect_type: Literal['ATE', 'ATT', 'direct', 'indirect', 'total']
9549:     point_estimate: float
9550:     posterior_mean: float
9551:     credible_interval_95: tuple[float, float]
9552:     probability_positive: float
9553:     probability_significant: float
9554:     mediating_paths: list[list[str]] = field(default_factory=list)
9555:     confounders_adjusted: list[str] = field(default_factory=list)
9556:
9557: @dataclass
9558: class CounterfactualScenario:
9559:     """Escenario contrafactual"""
9560:     intervention: dict[str, float]
9561:     predicted_outcomes: dict[str, tuple[float, float, float]]
9562:     probability_improvement: dict[str, float]
9563:     narrative: str
9564:
9565: @dataclass
9566: class ExtractedTable:
9567:     df: pd.DataFrame
9568:     page_number: int
9569:     table_type: str | None
9570:     extraction_method: Literal['camelot_lattice', 'camelot_stream', 'tabula', 'pdfplumber']
9571:     confidence_score: float
9572:     is_fragmented: bool = False
9573:     continuation_of: int | None = None
9574:
9575: @dataclass
9576: class FinancialIndicator:
9577:     source_text: str
9578:     amount: Decimal
9579:     currency: str
9580:     fiscal_year: int | None
9581:     funding_source: str
9582:     budget_category: str
9583:     execution_percentage: float | None
9584:     confidence_interval: tuple[float, float]
```

```
9585:     risk_level: float
9586:
9587: @dataclass
9588: class ResponsibleEntity:
9589:     name: str
9590:     entity_type: Literal['secretarÃ-a', 'oficina', 'direcciÃ³n', 'alcaldÃ-a', 'externo']
9591:     specificity_score: float
9592:     mentioned_count: int
9593:     associated_programs: list[str]
9594:     associated_indicators: list[str]
9595:     budget_allocated: Decimal | None
9596:
9597: @dataclass
9598: class QualityScore:
9599:     overall_score: float
9600:     financial_feasibility: float
9601:     indicator_quality: float
9602:     responsibility_clarity: float
9603:     temporal_consistency: float
9604:     pdet_alignment: float
9605:     causal_coherence: float
9606:     confidence_interval: tuple[float, float]
9607:     evidence: dict[str, Any]
9608:
9609: # =====
9610: # MOTOR PRINCIPAL
9611: # =====
9612:
9613: class PDET MunicipalPlanAnalyzer:
9614:     """Analizador de vanguardia para Planes de Desarrollo Municipal PDET"""
9615:
9616:     def __init__(self, use_gpu: bool = True, language: str = 'es', confidence_threshold: float = 0.7) -> None:
9617:         self.device = 'cuda' if use_gpu and torch.cuda.is_available() else 'cpu'
9618:         self.confidence_threshold = confidence_threshold
9619:         self.context = ColombianMunicipalContext()
9620:
9621:         print("δ\237\224$ Inicializando modelos de vanguardia...")
9622:
9623:         self.semantic_model = SentenceTransformer(
9624:             'sentence-transformers/paraphrase-multilingual-mpnet-base-v2',
9625:             device=self.device
9626:         )
9627:
9628:         # Delegate to factory for I/O operation
9629:         from farfan_pipeline.analysis.factory import load_spacy_model
9630:
9631:         try:
9632:             self.nlp = load_spacy_model("es_dep_news_trf")
9633:         except OSError:
9634:             raise RuntimeError(
9635:                 "Modelo SpaCy 'es_dep_news_trf' no instalado. "
9636:                 "Ejecuta: python -m spacy download es_dep_news_trf"
9637:             )
9638:
9639:         self.entity_classifier = pipeline(
9640:             "token-classification",
```

```

9641:         model="mrm8488/bert-spanish-cased-finetuned-ner",
9642:         device=0 if use_gpu else -1,
9643:         aggregation_strategy="simple"
9644:     )
9645:
9646:     self.tfidf = TfidfVectorizer(
9647:         max_features=1000,
9648:         ngram_range=(1, 3),
9649:         min_df=2,
9650:         stop_words=self._get_spanish_stopwords()
9651:     )
9652:
9653:     self.pdet_embeddings = {
9654:         pillar: self.semantic_model.encode(pillar, convert_to_tensor=False)
9655:         for pillar in self.context.PDET_PILLARS
9656:     }
9657:
9658:     print("â\234\205 Modelos inicializados correctamente\n")
9659:
9660: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._get_spanish_stopwords")
9661: def _get_spanish_stopwords(self) -> list[str]:
9662:     base_stopwords = spacy.lang.es.stop_words.STOP_WORDS
9663:     gov_stopwords = {
9664:         'artÃculo', 'decreto', 'mediante', 'conforme', 'respecto',
9665:         'acuerdo', 'resoluciÃ³n', 'ordenanza', 'literal', 'numeral'
9666:     }
9667:     return list(base_stopwords | gov_stopwords)
9668:
9669: # =====
9670: # EXTRACCIÃ\223N DE TABLAS
9671: # =====
9672:
9673: async def extract_tables(self, pdf_path: str) -> list[ExtractedTable]:
9674:     print("â\237\223\212 Iniciando extracciÃ³n avanzada de tablas...")
9675:     all_tables: list[ExtractedTable] = []
9676:     pdf_path_str = str(pdf_path)
9677:
9678:     # Camelot Lattice
9679:     try:
9680:         lattice_tables = camelot.read_pdf(
9681:             pdf_path_str, pages='all', flavor='lattice',
9682:             line_scale=40, joint_tol=10, edge_tol=50
9683:         )
9684:         for idx, table in enumerate(lattice_tables):
9685:             if table.parsing_report['accuracy'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._ge
t_spanish_stopwords", "auto_param_L342_54", 0.7):
9686:                 all_tables.append(ExtractedTable(
9687:                     df=self._clean_dataframe(table.df),
9688:                     page_number=table.page,
9689:                     table_type=None,
9690:                     extraction_method='camelot_lattice',
9691:                     confidence_score=table.parsing_report['accuracy']
9692:                 ))
9693:     except Exception as e:
9694:         print(f" â\232 i,\217 Camelot Lattice: {str(e)[:50]}")
9695:

```

```

9696:     # Camelot Stream
9697:     try:
9698:         stream_tables = camelot.read_pdf(
9699:             pdf_path_str, pages='all', flavor='stream',
9700:             edge_tol=500, row_tol=15, column_tol=10
9701:         )
9702:         for idx, table in enumerate(stream_tables):
9703:             if table.parsing_report['accuracy'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._get_top(spanish_stopwords", "auto_param_L360_54", 0.6):
9704:                 all_tables.append(ExtractedTable(
9705:                     df=self._clean_dataframe(table.df),
9706:                     page_number=table.page,
9707:                     table_type=None,
9708:                     extraction_method='camelot_stream',
9709:                     confidence_score=table.parsing_report['accuracy']
9710:                 ))
9711:             except Exception as e:
9712:                 print(f" \u232a \u2171 Camelot Stream: {str(e)[:50]}")
9713:
9714:     # Tabula
9715:     try:
9716:         tabula_tables = tabula.read_pdf(
9717:             pdf_path_str, pages='all', multiple_tables=True,
9718:             stream=True, guess=True, silent=True
9719:         )
9720:         for idx, df in enumerate(tabula_tables):
9721:             if not df.empty and len(df) > 2:
9722:                 all_tables.append(ExtractedTable(
9723:                     df=self._clean_dataframe(df),
9724:                     page_number=idx + 1,
9725:                     table_type=None,
9726:                     extraction_method='tabula',
9727:                     confidence_score = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._get_topwords", "confidence_score", 0.6) # Refactored
9728:                 ))
9729:             except Exception as e:
9730:                 print(f" \u232a \u2171 Tabula: {str(e)[:50]}")
9731:
9732:         unique_tables = self._deduplicate_tables(all_tables)
9733:         print(f"\u232a \u2020 {len(unique_tables)} tablas \u00f3nicas extra\u00e1das\n")
9734:
9735:         reconstructed = await self._reconstruct_fragmented_tables(unique_tables)
9736:         print(f"\u2327 \u2248 \u2227 {len(reconstructed)} tablas despu\u00e9s de reconstituci\u00f3n\n")
9737:
9738:         classified = self._classify_tables(reconstructed)
9739:         return classified
9740:
9741:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._clean_dataframe")
9742:     def _clean_dataframe(self, df: pd.DataFrame) -> pd.DataFrame:
9743:         if df.empty:
9744:             return df
9745:         df = df.dropna(how='all').reset_index(drop=True)
9746:         df = df.dropna(axis=1, how='all')
9747:
9748:         if len(df) > 0:
9749:             first_row = df.iloc[0].astype(str)

```

```

9750:         if self._is_likely_header(first_row):
9751:             df.columns = first_row.values
9752:             df = df.iloc[1:].reset_index(drop=True)
9753:
9754:         for col in df.columns:
9755:             df[col] = df[col].astype(str).str.strip()
9756:             df[col] = df[col].replace(['', 'nan', 'None'], np.nan)
9757:
9758:         return df
9759:
9760:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._is_likely_header")
9761:     def _is_likely_header(self, row: pd.Series, **kwargs) -> bool:
9762:         """
9763:             Determine if a DataFrame row is likely a header row based on linguistic analysis.
9764:
9765:             Args:
9766:                 row: pandas Series representing a row from a DataFrame
9767:                 **kwargs: Accepts additional keyword arguments for backward compatibility.
9768:                         These are ignored (e.g., pdf_path if mistakenly passed).
9769:
9770:             Returns:
9771:                 Boolean indicating whether the row appears to be a header
9772:
9773:             Note:
9774:                 This function only requires 'row' parameter. Any additional kwargs
9775:                 (like 'pdf_path') are silently ignored to maintain interface stability.
9776:             """
9777:             # Log warning if unexpected kwargs are passed
9778:             if kwargs:
9779:                 logger.warning(
9780:                     f"_is_likely_header received unexpected keyword arguments: {list(kwargs.keys())}. "
9781:                     "These will be ignored. Expected signature: _is_likely_header(self, row: pd.Series)"
9782:                 )
9783:
9784:             text = ' '.join(row.astype(str))
9785:             doc = self.nlp(text)
9786:             pos_counts = pd.Series([token.pos_ for token in doc]).value_counts()
9787:             noun_ratio = pos_counts.get('NOUN', 0) / max(len(doc), 1)
9788:             verb_ratio = pos_counts.get('VERB', 0) / max(len(doc), 1)
9789:             return noun_ratio > verb_ratio and len(text) < 200
9790:
9791:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._deduplicate_tables")
9792:     def _deduplicate_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
9793:         if len(tables) <= 1:
9794:             return tables
9795:
9796:         embeddings = []
9797:         for table in tables:
9798:             table_text = table.df.to_string()[:1000]
9799:             emb = self.semantic_model.encode(table_text, convert_to_tensor=True)
9800:             embeddings.append(emb)
9801:
9802:         similarities = util.cos_sim(torch.stack(embeddings), torch.stack(embeddings))
9803:
9804:         to_keep = []
9805:         seen = set()

```

```
9806:         for i, table in enumerate(tables):
9807:             if i in seen:
9808:                 continue
9809:             duplicates = (similarities[i] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables", "auto_param_L466_44", 0.85)).nonzero(as_tuple=True)[0].tolist()
9810:             best_idx = max(duplicates, key=lambda idx: tables[idx].confidence_score)
9811:             to_keep.append(tables[best_idx])
9812:             seen.update(duplicates)
9813:
9814:     return to_keep
9815:
9816:     async def _reconstruct_fragmented_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
9817:         if len(tables) < 2:
9818:             return tables
9819:
9820:         features = []
9821:         for table in tables:
9822:             col_structure = '|'.join(sorted(str(c)[:20] for c in table.df.columns))
9823:             dtypes = '|'.join(sorted(str(dt) for dt in table.df.dtypes))
9824:             content = table.df.to_string()[:500]
9825:             combined = f"{col_structure} {dtypes} {content}"
9826:             features.append(combined)
9827:
9828:         embeddings = self.semantic_model.encode(features, convert_to_tensor=False)
9829:         clustering = DBSCAN(eps=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._deduplicate_tables", "auto_param_L486_32", 0.3), min_samples=2, metric='cosine').fit(embeddings)
9830:
9831:         reconstructed = []
9832:         processed = set()
9833:         for cluster_id in set(clustering.labels_):
9834:             if cluster_id == -1:
9835:                 continue
9836:             cluster_indices = np.where(clustering.labels_ == cluster_id)[0]
9837:             if len(cluster_indices) > 1:
9838:                 sorted_indices = sorted(cluster_indices, key=lambda i: tables[i].page_number)
9839:                 dfs_to_concat = [tables[i].df for i in sorted_indices]
9840:                 merged_df = pd.concat(dfs_to_concat, ignore_index=True)
9841:                 main_table = tables[sorted_indices[0]]
9842:                 reconstructed.append(ExtractedTable(
9843:                     df=merged_df,
9844:                     page_number=main_table.page_number,
9845:                     table_type=main_table.table_type,
9846:                     extraction_method=main_table.extraction_method,
9847:                     confidence_score=np.mean([tables[i].confidence_score for i in sorted_indices]),
9848:                     is_fragmented=True,
9849:                     continuation_of=None
9850:                 ))
9851:                 processed.update(sorted_indices)
9852:
9853:         for i, table in enumerate(tables):
9854:             if i not in processed:
9855:                 reconstructed.append(table)
9856:
9857:     return reconstructed
9858:
9859:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._classify_tables")
```

```

9860:     def _classify_tables(self, tables: list[ExtractedTable]) -> list[ExtractedTable]:
9861:         classification_patterns = {
9862:             'presupuesto': ['presupuesto', 'recursos', 'millones', 'sgp', 'sgr', 'fuente', 'financiaciÃ³n'],
9863:             'indicadores': ['indicador', 'lÃnea base', 'meta', 'fÃ³rmula', 'unidad de medida', 'periodicidad'],
9864:             'cronograma': ['cronograma', 'actividad', 'mes', 'trimestre', 'aÃ±o', 'fecha'],
9865:             'responsables': ['responsable', 'secretarÃ¡a', 'direcciÃ³n', 'oficina', 'ejecutor'],
9866:             'diagnÃ³stico': ['diagnÃ³stico', 'problema', 'causa', 'efecto', 'situaciÃ³n actual'],
9867:             'pdet': ['pdet', 'iniciativa', 'pilar', 'patr', 'transformaciÃ³n regional']
9868:         }
9869:
9870:         for table in tables:
9871:             table_text = table.df.to_string().lower()
9872:             scores = {}
9873:             for table_type, keywords in classification_patterns.items():
9874:                 score = sum(1 for kw in keywords if kw in table_text)
9875:                 scores[table_type] = score
9876:
9877:             if max(scores.values()) > 0:
9878:                 table.table_type = max(scores, key=scores.get)
9879:
9880:         return tables
9881:
9882: # =====
9883: # ANÃLISIS FINANCIERO
9884: # =====
9885:
9886: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.analyze_financial_feasibility")
9887: def analyze_financial_feasibility(self, tables: list[ExtractedTable], text: str) -> dict[str, Any]:
9888:     print("δ\237\222° Analizando feasibility financiero...")
9889:
9890:     financial_indicators = self._extract_financial_amounts(text, tables)
9891:     funding_sources = self._analyze_funding_sources(financial_indicators, tables)
9892:     sustainability = self._assess_financial_sustainability(financial_indicators, funding_sources)
9893:     risk_assessment = self._bayesian_risk_inference(financial_indicators, funding_sources, sustainability)
9894:
9895:     return {
9896:         'total_budget': sum(ind.amount for ind in financial_indicators),
9897:         'financial_indicators': [self._indicator_to_dict(ind) for ind in financial_indicators],
9898:         'funding_sources': funding_sources,
9899:         'sustainability_score': sustainability,
9900:         'risk_assessment': risk_assessment,
9901:         'confidence': risk_assessment['confidence_interval']
9902:     }
9903:
9904: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._extract_financial_amounts")
9905: def _extract_financial_amounts(self, text: str, tables: list[ExtractedTable]) -> list[FinancialIndicator]:
9906:     patterns = [
9907:         r'\$*\s*(\d{1,3}(?:[,]\d{3})*(?:[,]\d{1,2})?)\s*millones?',
9908:         r'\$*\s*(\d{1,3}(?:[,]\d{3})*(?:[,]\d{1,2})?)\s*(?:mil\s+)?millones?',
9909:         r'\$*\s*(\d{1,3}(?:[,]\d{3})*(?:[,]\d{1,2})?)',
9910:         r'(\d{1,6})\s*SMMLV'
9911:     ]
9912:
9913:     indicators = []
9914:     for pattern in patterns:
9915:         for match in re.finditer(pattern, text, re.IGNORECASE):

```

```

9916:         amount_str = match.group(1).replace('.', '').replace(',', '.')
9917:         try:
9918:             amount = Decimal(amount_str)
9919:             if 'millon' in match.group(0).lower():
9920:                 amount *= Decimal('1000000')
9921:
9922:             context_start = max(0, match.start() - 200)
9923:             context_end = min(len(text), match.end() + 200)
9924:             context = text[context_start:context_end]
9925:
9926:             funding_source = self._identify_funding_source(context)
9927:             year_match = re.search(r'20\d{2}', context)
9928:             fiscal_year = int(year_match.group()) if year_match else None
9929:
9930:             indicators.append(FinancialIndicator(
9931:                 source_text=match.group(0),
9932:                 amount=amount,
9933:                 currency='COP',
9934:                 fiscal_year=fiscal_year,
9935:                 funding_source=funding_source,
9936:                 budget_category='',
9937:                 execution_percentage=None,
9938:                 confidence_interval=(ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._extract_financial_amounts", "auto_param_L595_45", 0.0), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._extract_financial_amounts", "auto_param_L595_50", 0.0)),
9939:                 risk_level = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._extract_financial_amounts", "risk_level", 0.0) # Refactored
9940:             ))
9941:             except (ValueError, Exception):
9942:                 continue
9943:
9944:             budget_tables = [t for t in tables if t.table_type == 'presupuesto']
9945:             for table in budget_tables:
9946:                 table_indicators = self._extract_from_budget_table(table.df)
9947:                 indicators.extend(table_indicators)
9948:
9949:             print(f" {len(indicators)} indicadores financieros extraídos")
9950:             return indicators
9951:
9952:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._identify_funding_source")
9953:     def _identify_funding_source(self, context: str) -> str:
9954:         sources = {
9955:             'SGP': ['sgp', 'sistema general de participaciones'],
9956:             'SGR': ['sgr', 'regalías', 'sistema general de regalías'],
9957:             'Recursos Propios': ['recursos propios', 'propios', 'ingresos corrientes'],
9958:             'Cofinanciación': ['cofinanciación', 'cofinanciado'],
9959:             'Crédito': ['crédito', 'préstamo', 'endeudamiento'],
9960:             'Cooperación': ['cooperación internacional', 'donación'],
9961:             'PDET': ['pdet', 'paz', 'transformación regional']
9962:         }
9963:
9964:         context_lower = context.lower()
9965:         for source_name, keywords in sources.items():
9966:             if any(kw in context_lower for kw in keywords):
9967:                 return source_name
9968:         return 'No especificada'

```

```
9969:  
9970:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_from_budget_table")  
9971:     def _extract_from_budget_table(self, df: pd.DataFrame) -> list[FinancialIndicator]:  
9972:         indicators = []  
9973:         amount_cols = [col for col in df.columns if any(  
9974:             kw in str(col).lower() for kw in ['monto', 'valor', 'presupuesto', 'recursos'])  
9975:         )]  
9976:         source_cols = [col for col in df.columns if any(  
9977:             kw in str(col).lower() for kw in ['fuente', 'financiaciÃ³n', 'origen'])  
9978:         )]  
9979:  
9980:         if not amount_cols:  
9981:             return indicators  
9982:  
9983:         amount_col = amount_cols[0]  
9984:         source_col = source_cols[0] if source_cols else None  
9985:  
9986:         for _, row in df.iterrows():  
9987:             try:  
9988:                 amount_str = str(row[amount_col])  
9989:                 amount_str = re.sub(r'^\d.,]', ',', amount_str)  
9990:                 if not amount_str:  
9991:                     continue  
9992:                 amount = Decimal(amount_str.replace('.', '').replace(',', '.'))  
9993:                 funding_source = str(row[source_col]) if source_col else 'No especificada'  
9994:  
9995:                 indicators.append(FinancialIndicator(  
9996:                     source_text=f"Tabla: {amount_str}",  
9997:                     amount=amount,  
9998:                     currency='COP',  
9999:                     fiscal_year=None,  
10000:                     funding_source=funding_source,  
10001:                     budget_category='',  
10002:                     execution_percentage=None,  
10003:                     confidence_interval=(ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_from_budget_table", "auto_param_L660_41", 0.0), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_from_budget_table", "auto_param_L660_46", 0.0)),  
10004:                     risk_level = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_from_budget_table", "risk_level", 0.0) # Refactored  
10005:                 ))  
10006:             except Exception:  
10007:                 continue  
10008:  
10009:             return indicators  
10010:  
10011:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._analyze_funding_sources")  
10012:     def _analyze_funding_sources(self, indicators: list[FinancialIndicator], tables: list[ExtractedTable]) -> dict[  
10013:         str, Any]:  
10014:         source_distribution = {}  
10015:         for ind in indicators:  
10016:             source = ind.funding_source  
10017:             source_distribution[source] = source_distribution.get(source, Decimal(0)) + ind.amount  
10018:  
10019:             total = sum(source_distribution.values())  
10020:             if total == 0:  
10021:                 return {'distribution': {}, 'diversity_index': ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._analyze_funding_sources")}
```

```

r._analyze_funding_sources", "auto_param_L678_59", 0.0)}
10022:
10023:     proportions = [float(amount / total) for amount in source_distribution.values()]
10024:     diversity = -sum(p * np.log(p) if p > 0 else 0 for p in proportions)
10025:
10026:     return {
10027:         'distribution': {k: float(v) for k, v in source_distribution.items()},
10028:         'diversity_index': float(diversity),
10029:         'max_diversity': np.log(len(source_distribution)),
10030:         'dependency_risk': ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._analyze_funding_sources",
"auto_param_L687_31", 1.0) - (diversity / np.log(max(len(source_distribution), 2)))
10031:     }
10032:
10033:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._assess_financial_sustainability")
10034:     def _assess_financial_sustainability(self, indicators: list[FinancialIndicator],
10035:                                         funding_sources: dict[str, Any]) -> float:
10036:         if not indicators:
10037:             return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._assess_financial_sustainability", "au
to_param_L694_19", 0.0)
10038:
10039:         diversity_score = min(funding_sources.get('diversity_index', 0) / funding_sources.get('max_diversity', 1), ParameterLoaderV2.get("farfan_core.analys
is.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L696_115", 1.0))
10040:
10041:         distribution = funding_sources.get('distribution', {})
10042:         total = sum(distribution.values())
10043:         own_resources = distribution.get('Recursos Propios', 0) / total if total > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_
tablas.PDET MunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L700_90", 0.0)
10044:         pdet_dependency = distribution.get('PDET', 0) / total if total > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDE
TMunicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L701_80", 0.0)
10045:         pdet_risk = min(pdet_dependency * 2, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._assess_fina
ncial_sustainability", "auto_param_L702_45", 1.0))
10046:
10047:         sustainability = (diversity_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._assess_finan
cial_sustainability", "auto_param_L704_44", 0.3) + own_resources * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyz
er._assess_financial_sustainability", "auto_param_L704_66", 0.4) + (1 - pdet_risk) * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET M
unicipalPlanAnalyzer._assess_financial_sustainability", "auto_param_L704_90", 0.3))
10048:         return float(sustainability)
10049:
10050:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._bayesian_risk_inference")
10051:     def _bayesian_risk_inference(self, indicators: list[FinancialIndicator], funding_sources: dict[str, Any],
10052:                                 sustainability: float) -> dict[str, Any]:
10053:         print(" \u03d1\237\216\u00b2 Ejecutando inferencia bayesiana...")
10054:
10055:         observed_data = {
10056:             'n_indicators': len(indicators),
10057:             'diversity': funding_sources.get('diversity_index', 0),
10058:             'sustainability': sustainability,
10059:             'dependency': funding_sources.get('dependency_risk', ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanA
nalyzer._bayesian_risk_inference", "auto_param_L716_65", 0.5))
10060:         }
10061:
10062:         with pm.Model():
10063:             base_risk = pm.Beta('base_risk', alpha=2, beta=5)
10064:             diversity_effect = pm.Normal('diversity_effect', mu=-ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanA
nalyzer._bayesian_risk_inference", "auto_param_L721_65", 0.3), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnal
yzer._bayesian_risk_inference", "auto_param_L721_76", 0.1))

```

```

10065:         sustainability_effect = pm.Normal('sustainability_effect', mu=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L722_75", 0.4), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L722_86", 0.1))
10066:         dependency_effect = pm.Normal('dependency_effect', mu=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L723_66", 0.5), sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._bayesian_risk_inference", "auto_param_L723_77", 0.15))
10067:
10068:     pm.Deterministic(
10069:         'risk',
10070:         pm.math.sigmoid(
10071:             pm.math.log(base_risk / (1 - base_risk)) +
10072:             diversity_effect * observed_data['diversity'] +
10073:             sustainability_effect * observed_data['sustainability'] +
10074:             dependency_effect * observed_data['dependency']
10075:         )
10076:     )
10077:
10078:     trace = pm.sample(2000, tune=1000, cores=1, return_inferencedata=True, progressbar=False)
10079:
10080:     risk_samples = trace.posterior['risk'].values.flatten()
10081:     risk_mean = float(np.mean(risk_samples))
10082:     risk_ci = tuple(float(x) for x in np.percentile(risk_samples, [2.5, 97.5]))
10083:
10084:     print(f" \u2192 Riesgo estimado: {risk_mean:.3f} CI95%: {risk_ci}")
10085:
10086:     return {
10087:         'risk_score': risk_mean,
10088:         'confidence_interval': risk_ci,
10089:         'interpretation': self._interpret_risk(risk_mean),
10090:         'posterior_samples': risk_samples.tolist()
10091:     }
10092:
10093:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_risk")
10094:     def _interpret_risk(self, risk: float) -> str:
10095:         if risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_risk", "auto_param_L752_18", 0.2):
10096:             return "Riesgo bajo - Plan financieramente robusto"
10097:         elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_risk", "auto_param_L754_20", 0.4):
10098:             return "Riesgo moderado-bajo - Sostenibilidad probable"
10099:         elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_risk", "auto_param_L756_20", 0.6):
10100:             return "Riesgo moderado - Requiere monitoreo"
10101:         elif risk < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_risk", "auto_param_L758_20", 0.8):
10102:             return "Riesgo alto - Vulnerabilidades significativas"
10103:         else:
10104:             return "Riesgo cr\u00e1tico - Inviabilidad financiera probable"
10105:
10106:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._indicator_to_dict")
10107:     def _indicator_to_dict(self, ind: FinancialIndicator) -> dict[str, Any]:
10108:         return {
10109:             'source_text': ind.source_text,
10110:             'amount': float(ind.amount),
10111:             'currency': ind.currency,
10112:             'fiscal_year': ind.fiscal_year,

```

```

10113:         'funding_source': ind.funding_source,
10114:         'risk_level': ind.risk_level
10115:     }
10116:
10117:     # =====
10118:     # IDENTIFICACIÃ“N DE RESPONSABLES
10119:     # =====
10120:
10121:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.identify_responsible_entities")
10122:     def identify_responsible_entities(self, text: str, tables: list[ExtractedTable]) -> list[ResponsibleEntity]:
10123:         print("Ã“237\221 Identificando entidades responsables...")
10124:
10125:         entities_ner = self._extract_entities_ner(text)
10126:         entities_syntax = self._extract_entities_syntax(text)
10127:         entities_tables = self._extract_from_responsibility_tables(tables)
10128:
10129:         all_entities = entities_ner + entities_syntax + entities_tables
10130:         unique_entities = self._consolidate_entities(all_entities)
10131:         scored_entities = self._score_entity_specificity(unique_entities, text)
10132:
10133:         print(f" {len(scored_entities)} entidades responsables identificadas")
10134:         return sorted(scored_entities, key=lambda x: x.specificity_score, reverse=True)
10135:
10136:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_entities_ner")
10137:     def _extract_entities_ner(self, text: str) -> list[ResponsibleEntity]:
10138:         entities = []
10139:         max_length = 512
10140:         words = text.split()
10141:         chunks = [' '.join(words[i:i + max_length]) for i in range(0, len(words), max_length)]
10142:
10143:         for chunk in chunks[:10]:
10144:             try:
10145:                 ner_results = self.entity_classifier(chunk)
10146:                 for entity in ner_results:
10147:                     if entity['entity_group'] in ['ORG', 'PER'] and entity['score'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_entities_ner", "auto_param_L804_86", 0.7):
10148:                         entities.append(ResponsibleEntity(
10149:                             name=entity['word'],
10150:                             entity_type='secretarÃ–a',
10151:                             specificity_score=entity['score'],
10152:                             mentioned_count=1,
10153:                             associated_programs=[],
10154:                             associated_indicators=[],
10155:                             budget_allocated=None
10156:                         ))
10157:             except Exception:
10158:                 continue
10159:
10160:         return entities
10161:
10162:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_entities_syntax")
10163:     def _extract_entities_syntax(self, text: str) -> list[ResponsibleEntity]:
10164:         entities = []
10165:         responsibility_patterns = [
10166:             r'(?:(?:responsable|ejecutor|encargado|a\s+cargo)\s+([A-ZÃ“\201-\Ã“\232][^.\n]{10,100}))',
10167:             r'(?:(?:secretarÃ–i|a|direcciÃ³n|oficina)\s+(?:de\s+)?([A-ZÃ“\201-\Ã“\232][^.\n]{5,80}))',

```

```

10168:         r'([A-ZÀ\201-\Ã\232][^.\n]{10,100})\s+(?:ser[Ã;a]|estar[Ã;a]|tendr[Ã;a])\s+(?:responsable|a cargo)'
10169:     ]
10170:
10171:     for pattern in responsibility_patterns:
10172:         for match in re.findall(pattern, text, re.MULTILINE):
10173:             name = match.group(1).strip()
10174:             if len(name) < 10 or len(name) > 150:
10175:                 continue
10176:
10177:             entity_type = self._classify_entity_type(name)
10178:             entities.append(ResponsibleEntity(
10179:                 name=name,
10180:                 entity_type=entity_type,
10181:                 specificity_score=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._extract_entities_s
yntax", "auto_param_L838_38", 0.6),
10182:                 mentioned_count=1,
10183:                 associated_programs=[],
10184:                 associated_indicators=[],
10185:                 budget_allocated=None
10186:             ))
10187:
10188:     return entities
10189:
10190: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._classify_entity_type")
10191: def _classify_entity_type(self, name: str) -> str:
10192:     name_lower = name.lower()
10193:     if 'secretarÃ-a' in name_lower or 'secretaria' in name_lower:
10194:         return 'secretarÃ-a'
10195:     elif 'direcciÃ³n' in name_lower:
10196:         return 'direcciÃ³n'
10197:     elif 'oficina' in name_lower:
10198:         return 'oficina'
10199:     elif 'alcaldÃ-a' in name_lower or 'alcalde' in name_lower:
10200:         return 'alcaldÃ-a'
10201:     else:
10202:         return 'externo'
10203:
10204: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._extract_from_responsibility_tables")
10205: def _extract_from_responsibility_tables(self, tables: list[ExtractedTable]) -> list[ResponsibleEntity]:
10206:     entities = []
10207:     resp_tables = [t for t in tables if t.table_type == 'responsables']
10208:
10209:     for table in resp_tables:
10210:         df = table.df
10211:         resp_cols = [col for col in df.columns if any(
10212:             kw in str(col).lower() for kw in ['responsable', 'ejecutor', 'encargado']
10213:         )]
10214:
10215:         if not resp_cols:
10216:             continue
10217:
10218:         resp_col = resp_cols[0]
10219:         for value in df[resp_col].dropna().unique():
10220:             name = str(value).strip()
10221:             if len(name) < 5:
10222:                 continue

```

```
10223:             entities.append(ResponsibleEntity(
10224:                 name=name,
10225:                 entity_type=self._classify_entity_type(name),
10226:                 specificity_score=ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_from_respo
nsibility_tables", "auto_param_L884_38", 0.8),
10227:                 mentioned_count=1,
10228:                 associated_programs=[],
10229:                 associated_indicators=[],
10230:                 budget_allocated=None
10231:             ))
10232:
10233:
10234:     return entities
10235:
10236: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._consolidate_entities")
10237: def _consolidate_entities(self, entities: list[ResponsibleEntity]) -> list[ResponsibleEntity]:
10238:     if not entities:
10239:         return []
10240:
10241:     names = [e.name for e in entities]
10242:     embeddings = self.semantic_model.encode(names, convert_to_tensor=True)
10243:
10244:     similarity_threshold = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._consolidate_entities", "s
imilarity_threshold", 0.85) # Refactored
10245:     clustering = AgglomerativeClustering(
10246:         n_clusters=None,
10247:         distance_threshold=1 - similarity_threshold,
10248:         metric='cosine',
10249:         linkage='average'
10250:     )
10251:     labels = clustering.fit_predict(embeddings.cpu().numpy())
10252:
10253:     consolidated = []
10254:     for cluster_id in set(labels):
10255:         cluster_entities = [e for i, e in enumerate(entities) if labels[i] == cluster_id]
10256:         best_entity = max(cluster_entities, key=lambda e: (len(e.name), e.specification_score, e.mentioned_count))
10257:         total_mentions = sum(e.mentioned_count for e in cluster_entities)
10258:
10259:         consolidated.append(ResponsibleEntity(
10260:             name=best_entity.name,
10261:             entity_type=best_entity.entity_type,
10262:             specification_score=best_entity.specification_score,
10263:             mentioned_count=total_mentions,
10264:             associated_programs=best_entity.associated_programs,
10265:             associated_indicators=best_entity.associated_indicators,
10266:             budget_allocated=best_entity.budget_allocated
10267:         ))
10268:
10269:     return consolidated
10270:
10271: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_entity_specificity")
10272: def _score_entity_specificity(self, entities: list[ResponsibleEntity], full_text: str) -> list[ResponsibleEntity]:
10273:     scored = []
10274:     for entity in entities:
10275:         doc = self.nlp(entity.name)
10276:
```

```

10277:         length_score = min(len(entity.name.split()) / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L934_62", 1.0))
10278:         propn_count = sum(1 for token in doc if token.pos_ == 'PROPN')
10279:         propn_score = min(propn_count / 3, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L936_47", 1.0))
10280:
10281:         institutional_words = ['secretarÃ-a', 'direcciÃ³n', 'oficina', 'departamento', 'coordinaciÃ³n', 'gerencia',
10282:                               'subdirecciÃ³n']
10283:         inst_score = float(any(word in entity.name.lower() for word in institutional_words))
10284:         mention_score = min(entity.mentioned_count / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L941_61", 1.0))
10285:
10286:         final_score = (length_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L943_42", 0.2) + propn_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L943_62", 0.3) + inst_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L943_81", 0.3) + mention_score * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._score_entity_specificity", "auto_param_L943_103", 0.2))
10287:
10288:         entity.specificity_score = final_score
10289:         scored.append(entity)
10290:
10291:     return scored
10292:
10293: # =====
10294: # INFERENCIA CAUSAL - DAG CONSTRUCTION
10295: # =====
10296:
10297: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer.construct_causal_dag")
10298: def construct_causal_dag(self, text: str, tables: list[ExtractedTable],
10299:                           financial_analysis: dict[str, Any]) -> CausalDAG:
10300:     print("δ\237\224\227 Construyendo grafo causal (DAG)...")
10301:
10302:     nodes = self._identify_causal_nodes(text, tables, financial_analysis)
10303:     print(f" \u234\223 {len(nodes)} nodos causales identificados")
10304:
10305:     edges = self._identify_causal_edges(text, nodes)
10306:     print(f" \u234\223 {len(edges)} relaciones causales detectadas")
10307:
10308:     G = nx.DiGraph()
10309:     for node_name, node in nodes.items():
10310:         G.add_node(node_name, **{
10311:             'type': node.node_type,
10312:             'budget': float(node.associated_budget) if node.associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer.construct_causal_dag", "auto_param_L969_87", 0.0),
10313:             'evidence': node.evidence_strength
10314:         })
10315:
10316:         for edge in edges:
10317:             if edge.probability > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer.construct_causal_dag",
10318: "auto_param_L974_34", 0.3):
10319:                 G.add_edge(edge.source, edge.target, **{
10320:                     'type': edge.edge_type,
10321:                     'mechanism': edge.mechanism,
10322:                     'probability': edge.probability
10323:                 })

```



```

10378:         mediator_mentions = self._find_mediator_mentions(text, mediator)
10379:         if len(mediator_mentions) > 0:
10380:             nodes[mediator] = CausalNode(
10381:                 name=mediator,
10382:                 node_type='mediator',
10383:                 embedding=self.semantic_model.encode(mediator, convert_to_tensor=False),
10384:                 associated_budget=None,
10385:                 temporal_lag=0,
10386:                 evidence_strength=min(len(mediator_mentions) / 2, ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDET Municipal
10387: ipalPlanAnalyzer._identify_causal_nodes", "auto_param_L1043_74", 1.0))
10388:             )
10389:         return nodes
10390:
10391:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._find_semantic_mentions")
10392:     def _find_semantic_mentions(self, text: str, concept: str, concept_embedding: np.ndarray) -> list[str]:
10393:         sentences = [s.text for s in self.nlp(text[:50000]).sents]
10394:
10395:         mentions = []
10396:         for sentence in sentences:
10397:             if len(sentence.split()) < 5:
10398:                 continue
10399:
10400:             sent_embedding = self.semantic_model.encode(sentence, convert_to_tensor=False)
10401:             similarity = np.dot(concept_embedding, sent_embedding) / (
10402:                 np.linalg.norm(concept_embedding) * np.linalg.norm(sent_embedding)
10403:             )
10404:
10405:             if similarity > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._find_semantic_mentions", "au
10406: to_param_L1062_28", 0.5):
10407:                 mentions.append(sentence)
10408:
10409:         return mentions
10410:
10411:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._find_outcome_mentions")
10412:     def _find_outcome_mentions(self, text: str, outcome: str) -> list[str]:
10413:         outcome_keywords = {
10414:             'seguridad_juridica': ['seguridad jurídica', 'formalización', 'títulos', 'propiedad'],
10415:             'reducción_conflictos_tienda': ['conflicto', 'tierra', 'disputa', 'territorial'],
10416:             'conectividad': ['conectividad', 'vías', 'acceso', 'transporte'],
10417:             'productividad_agrícola': ['productividad', 'agrícola', 'producción', 'rendimiento'],
10418:             'mortalidad_infantil': ['mortalidad infantil', 'niños', 'salud infantil'],
10419:             'esperanza_vida': ['esperanza de vida', 'longevidad', 'salud'],
10420:             'cobertura_educativa': ['cobertura educativa', 'acceso educación', 'matrícula'],
10421:             'calidad_educativa': ['calidad educativa', 'aprendizaje', 'pruebas saber'],
10422:             'deficit_habitacional': ['dáficit habitacional', 'vivienda', 'hogares'],
10423:             'enfermedades_hidráticas': ['enfermedades hídricas', 'agua potable', 'saneamiento'],
10424:             'ingreso_rural': ['ingreso rural', 'pobreza rural', 'economía campesina'],
10425:             'empleo_rural': ['empleo rural', 'trabajo campo', 'ocupación'],
10426:             'seguridad_alimentaria': ['seguridad alimentaria', 'hambre', 'nutrición'],
10427:             'nutrición_infantil': ['nutrición infantil', 'desnutrición', 'alimentación niños'],
10428:             'cohesión_social': ['cohesión social', 'tejido social', 'comunidad'],
10429:             'confianza_institucional': ['confianza', 'instituciones', 'legitimidad']
10430:
10431:         keywords = outcome_keywords.get(outcome, [outcome])

```

```

10432:     text_lower = text.lower()
10433:
10434:     mentions = []
10435:     for keyword in keywords:
10436:         if keyword in text_lower:
10437:             pattern = f'.{{0,100}}{{re.escape(keyword)}}.{{0,100}}'
10438:             matches = re.finditer(pattern, text_lower, re.IGNORECASE)
10439:             mentions.extend([m.group() for m in matches])
10440:
10441:     return mentions[:10]
10442:
10443: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._find_mediator_mentions")
10444: def _find_mediator_mentions(self, text: str, mediator: str) -> list[str]:
10445:     mediator_patterns = {
10446:         'formalizacion': ['formalizaciÃ³n', 'titulaciÃ³n', 'escrituras'],
10447:         'acceso_justicia': ['acceso justicia', 'juzgados', 'defensorÃ-a'],
10448:         'vias_terciarias': ['vÃ-as terciarias', 'caminos', 'carreteras'],
10449:         'distritos_riego': ['distritos riego', 'irrigaciÃ³n', 'agua agrÃ-a-cola'],
10450:         'cobertura_salud': ['cobertura salud', 'eps', 'atenciÃ³n mÃ©dica'],
10451:         'infraestructura_salud': ['hospital', 'centro salud', 'puesto salud'],
10452:         'infraestructura_escolar': ['escuela', 'colegio', 'infraestructura educativa'],
10453:         'docentes_calificados': ['docentes', 'maestros', 'profesores'],
10454:         'cobertura_acueducto': ['acueducto', 'agua potable', 'tuberÃ-a'],
10455:         'viviendas_dignas': ['vivienda digna', 'casa', 'hogar'],
10456:         'credito_rural': ['crÃ©dito rural', 'financiamiento', 'banco agrario'],
10457:         'asistencia_tecnica': ['asistencia tÃ©cnica', 'extensiÃ³n rural', 'asesorÃ-a'],
10458:         'produccion_local': ['producciÃ³n local', 'cultivos', 'agricultura'],
10459:         'acceso_alimentos': ['acceso alimentos', 'mercado', 'distribuciÃ³n'],
10460:         'espacios_participacion': ['participaciÃ³n', 'comitÃ©s', 'juntas'],
10461:         'justicia_transicional': ['justicia transicional', 'vÃ-ctimas', 'reparaciÃ³n']
10462:     }
10463:
10464:     patterns = mediator_patterns.get(mediator, [mediator])
10465:     text_lower = text.lower()
10466:
10467:     mentions = []
10468:     for pattern in patterns:
10469:         if pattern in text_lower:
10470:             matches = re.finditer(f'.{{0,80}}{{re.escape(pattern)}}.{{0,80}}', text_lower)
10471:             mentions.extend([m.group() for m in matches])
10472:
10473:     return mentions[:8]
10474:
10475: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._extract_budget_for_pillar")
10476: def _extract_budget_for_pillar(self, pillar: str, text: str, financial_analysis: dict[str, Any]) -> Decimal | None:
10477:     pillar_lower = pillar.lower()
10478:
10479:     for indicator in financial_analysis.get('financial_indicators', []):
10480:         try:
10481:             source_start = text.lower().find(indicator['source_text'].lower())
10482:             if source_start == -1:
10483:                 continue
10484:
10485:             context_start = max(0, source_start - 500)
10486:             context_end = min(len(text), source_start + 500)
10487:             context = text[context_start:context_end].lower()

```

```

10488:
10489:         if pillar_lower in context:
10490:             return Decimal(str(indicator['amount']))
10491:         except Exception:
10492:             continue
10493:
10494:     return None
10495:
10496: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges")
10497: def _identify_causal_edges(self, text: str, nodes: dict[str, CausalNode]) -> list[CausalEdge]:
10498:     edges = []
10499:
10500:     for pillar, theory in self.context.PDET_THEORY_OF_CHANGE.items():
10501:         if pillar not in nodes:
10502:             continue
10503:
10504:         for mediator in theory['mediators']:
10505:             if mediator in nodes:
10506:                 edges.append(CausalEdge(
10507:                     source=pillar,
10508:                     target=mediator,
10509:                     edge_type='direct',
10510:                     mechanism="Mecanismo segÃ³n teorÃ-a PDET",
10511:                     probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.8) # Refactored
10512:                 ))
10513:
10514:         for outcome in theory['outcomes']:
10515:             if outcome in nodes:
10516:                 for mediator in theory['mediators']:
10517:                     if mediator in nodes:
10518:                         edges.append(CausalEdge(
10519:                             source=mediator,
10520:                             target=outcome,
10521:                             edge_type='mediated',
10522:                             mechanism=f"Mediado por {mediator}",
10523:                             probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.7) # Refactored
10524:                         ))
10525:
10526:             causal_patterns = [
10527:                 r'(.+)\s+(?:genera|produce|causa|lleva a|resulta en|permite)\s+(.+?)\[\.\,\]', 'direct'),
10528:                 r'(.+)\s+mediante\s+(.+) \s+(?:se logra|alcanza|obtiene)\s+', 'mediated'),
10529:                 r'para\s+(?:lograr|alcanzar)\s+(.+) \s+se requiere\s+(.+) \[\.\,\]', 'direct')
10530:             ]
10531:
10532:             for pattern, edge_type in causal_patterns:
10533:                 for match in re.finditer(pattern, text[:30000], re.IGNORECASE):
10534:                     source_text = match.group(1).strip()
10535:                     target_text = match.group(2).strip() if match.lastindex >= 2 else ""
10536:
10537:                     source_node = self._match_text_to_node(source_text, nodes)
10538:                     target_node = self._match_text_to_node(target_text, nodes)
10539:
10540:                     if source_node and target_node and source_node != target_node:
10541:                         existing = next((e for e in edges if e.source == source_node and e.target == target_node), None)

```

```

10542:
10543:             if existing:
10544:                 existing.probability = min(existing.probability + ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges", "auto_param_L1201_74", 0.2), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges", "auto_param_L1201_79", 1.0))
10545:                 existing.evidence_quotes.append(match.group(0)[:200])
10546:             else:
10547:                 edges.append(CausalEdge(
10548:                     source=source_node,
10549:                     target=target_node,
10550:                     edge_type=edge_type,
10551:                     mechanism=match.group(0)[:200],
10552:                     evidence_quotes=[match.group(0)[:200]],
10553:                     probability = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_causal_edges", "probability", 0.6) # Refactored
10554:                 ))
10555:
10556:             edges = self._refine_edge_probabilities(edges, text, nodes)
10557:
10558:         return edges
10559:
10560:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._match_text_to_node")
10561:     def _match_text_to_node(self, text: str, nodes: dict[str, CausalNode]) -> str | None:
10562:         if len(text) < 5:
10563:             return None
10564:
10565:         text_embedding = self.semantic_model.encode(text, convert_to_tensor=False)
10566:
10567:         best_match = None
10568:         best_similarity = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._match_text_to_node", "best_similarity", 0.0) # Refactored
10569:
10570:         for node_name, node in nodes.items():
10571:             if node.embedding is None:
10572:                 continue
10573:
10574:             similarity = np.dot(text_embedding, node.embedding) / (
10575:                 np.linalg.norm(text_embedding) * np.linalg.norm(node.embedding) + 1e-10
10576:             )
10577:
10578:             if similarity > best_similarity and similarity > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._match_text_to_node", "auto_param_L1235_61", 0.4):
10579:                 best_similarity = similarity
10580:                 best_match = node_name
10581:
10582:         return best_match
10583:
10584:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._refine_edge_probabilities")
10585:     def _refine_edge_probabilities(self, edges: list[CausalEdge], text: str, nodes: dict[str, CausalNode]) -> list[
10586:         CausalEdge]:
10587:         text_lower = text.lower()
10588:
10589:         for edge in edges:
10590:             text_lower.count(edge.source[:30].lower())
10591:             text_lower.count(edge.target[:30].lower())
10592:

```

```

10593:     cooccurrence_count = 0
10594:     positions_source = [m.start() for m in re.finditer(re.escape(edge.source[:30].lower()), text_lower)]
10595:     positions_target = [m.start() for m in re.finditer(re.escape(edge.target[:30].lower()), text_lower)]
10596:
10597:     for pos_s in positions_source:
10598:         for pos_t in positions_target:
10599:             if abs(pos_s - pos_t) < 500:
10600:                 cooccurrence_count += 1
10601:
10602:             if cooccurrence_count > 0:
10603:                 boost = min(cooccurrence_count * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1260_49", 0.1), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1260_54", 0.3))
10604:                 edge.probability = min(edge.probability + boost, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._refine_edge_probabilities", "auto_param_L1261_65", 1.0))
10605:
10606:             return edges
10607:
10608: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._break_cycles")
10609: def _break_cycles(self, G: nx.DiGraph) -> nx.DiGraph:
10610:     while not nx.is_directed_acyclic_graph(G):
10611:         try:
10612:             cycle = nx.find_cycle(G)
10613:             weakest_edge = min(cycle, key=lambda e: G[e[0]][e[1]].get('probability', ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._break_cycles", "auto_param_L1270_89", 0.5)))
10614:             G.remove_edge(weakest_edge[0], weakest_edge[1])
10615:         except nx.NetworkXNoCycle:
10616:             break
10617:
10618:     return G
10619:
10620: # =====
10621: # ESTIMACIÃ\223N BAYESIANA DE EFECTOS CAUSALES
10622: # =====
10623:
10624: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.estimate_causal_effects")
10625: def estimate_causal_effects(self, dag: CausalDAG, text: str, financial_analysis: dict[str, Any]) -> list[CausalEffect]:
10626:     CausalEffect:
10627:     print("ð\237\223\210 Estimando efectos causales bayesianos...")
10628:
10629:     effects = []
10630:     G = dag.graph
10631:
10632:     for source in dag.nodes:
10633:         if dag.nodes[source].node_type != 'pilar':
10634:             continue
10635:
10636:         reachable_outcomes = [
10637:             node for node, data in G.nodes(data=True)
10638:             if data.get('type') == 'outcome' and nx.has_path(G, source, node)
10639:         ]
10640:
10641:         for outcome in reachable_outcomes:
10642:             effect = self._estimate_effect_bayesian(source, outcome, dag, financial_analysis)
10643:
10644:             if effect:

```

```

10645:         effects.append(effect)
10646:
10647:     print(f" {len(effects)} efectos causales estimados")
10648:     return effects
10649:
10650: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian")
10651: def _estimate_effect_bayesian(self, treatment: str, outcome: str, dag: CausalDAG,
10652:                               financial_analysis: dict[str, Any]) -> CausalEffect | None:
10653:     G = dag.graph
10654:     try:
10655:         all_paths = list(nx.all_simple_paths(G, treatment, outcome, cutoff=4))
10656:     except (nx.NetworkXNoPath, nx.NodeNotFound):
10657:         return None
10658:
10659:     if not all_paths:
10660:         return None
10661:
10662:     [p for p in all_paths if len(p) == 2]
10663:     indirect_paths = [p for p in all_paths if len(p) > 2]
10664:
10665:     confounders = self._identify_confounders(treatment, outcome, dag)
10666:
10667:     treatment_node = dag.nodes[treatment]
10668:     budget_value = float(treatment_node.associated_budget) if treatment_node.associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1325_104", 0.0)
10669:
10670:     with pm.Model():
10671:         prior_mean, prior_sd = self._get_prior_effect(treatment, outcome)
10672:
10673:         direct_effect = pm.StudentT('direct_effect', nu=3, mu=prior_mean, sigma=prior_sd)
10674:
10675:         indirect_effects = []
10676:         for path in indirect_paths[:3]:
10677:             path_name = '->'.join([p[:15] for p in path])
10678:             indirect_eff = pm.Normal(f'indirect_{path_name}', mu=prior_mean * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1335_82", 0.5), sigma=prior_sd * 1.5)
10679:             indirect_effects.append(indirect_eff)
10680:
10681:         if budget_value > 0:
10682:             budget_adjustment = pm.Deterministic('budget_adjustment', pm.math.logip(budget_value / 1e9))
10683:             adjusted_direct = direct_effect * (1 + budget_adjustment * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1340_75", 0.1))
10684:         else:
10685:             adjusted_direct = direct_effect
10686:
10687:         if indirect_effects:
10688:             total_effect = pm.Deterministic('total_effect', adjusted_direct + pm.math.sum(indirect_effects))
10689:         else:
10690:             total_effect = pm.Deterministic('total_effect', adjusted_direct)
10691:
10692:         evidence_strength = treatment_node.evidence_strength * dag.nodes[outcome].evidence_strength
10693:         obs_noise = pm.HalfNormal('obs_noise', sigma=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1350_57", 0.5))
10694:
10695:         pm.Normal('pseudo_obs', mu=total_effect, sigma=obs_noise,
10696:                  observed=np.array([evidence_strength * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunic

```

```

ipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1353_74", 0.5)))))

10697:
10698:     trace = pm.sample(1500, tune=800, cores=1, return_inferencedata=True, progressbar=False, target_accept=ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1355_115", 0.9))
10699:
10700:     total_samples = trace.posterior['total_effect'].values.flatten()
10701:     trace.posterior['direct_effect'].values.flatten()
10702:
10703:     total_mean = float(np.mean(total_samples))
10704:     total_ci = tuple(float(x) for x in np.percentile(total_samples, [2.5, 97.5]))
10705:     prob_positive = float(np.mean(total_samples > 0))
10706:     prob_significant = float(np.mean(np.abs(total_samples) > ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._estimate_effect_bayesian", "auto_param_L1363_65", 0.1)))
10707:
10708:     return CausalEffect(
10709:         treatment=treatment,
10710:         outcome=outcome,
10711:         effect_type='total',
10712:         point_estimate=float(np.median(total_samples)),
10713:         posterior_mean=total_mean,
10714:         credible_interval_95=total_ci,
10715:         probability_positive=prob_positive,
10716:         probability_significant=prob_significant,
10717:         mediating_paths=indirect_paths,
10718:         confounders_adjusted=confounders
10719:     )
10720:
10721: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect")
10722: def _get_prior_effect(self, treatment: str, outcome: str) -> tuple[float, float]:
10723:     """
10724:     Priors informados basados en meta-análisis de programas PDET
10725:     Referencia: Cinelli et al. (2022) - Sensitivity Analysis for Causal Inference
10726:     """
10727:     effect_priors = {
10728:         ('Infraestructura y adecuación de tierras', 'productividad_agricola'): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1385_84", 0.35), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1385_90", 0.15)),
10729:         ('Salud rural', 'mortalidad_infantil'): (-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1386_54", 0.28), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1386_60", 0.12)),
10730:         ('Educación rural y primera infancia', 'cobertura_educativa'): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1387_76", 0.42), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1387_82", 0.18)),
10731:         ('Vivienda, agua potable y saneamiento básico', 'enfermedades_hidricas'): (-ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1388_88", 0.33), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1388_94", 0.14)),
10732:         ('Reactivación económica y producción agropecuaria', 'ingreso_rural'): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1389_84", 0.29), ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1389_90", 0.16)),
10733:         ('Sistema para la garantía progresiva del derecho a la alimentación', 'seguridad_alimentaria'): (ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1390_109", 0.38),
10734:                                         ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1391_108", 0.17)),
10735:     }
10736:
10737:     if (treatment, outcome) in effect_priors:

```

```

10738:         return effect_priors[(treatment, outcome)]
10739:
10740:     return (ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1397_16"
10741: , 0.2), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._get_prior_effect", "auto_param_L1397_21", 0.25))
10742: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._identify_confounders")
10743: def _identify_confounders(self, treatment: str, outcome: str, dag: CausalDAG) -> list[str]:
10744: """
10745: Identifica confounders usando d-separation (Pearl, 2009)
10746: """
10747: G = dag.graph
10748: confounders = []
10749:
10750: for node in G.nodes():
10751:     if node in (treatment, outcome):
10752:         continue
10753:
10754:     if G.has_edge(node, treatment) and G.has_edge(node, outcome):
10755:         confounders.append(node)
10756:
10757: return confounders
10758:
10759: # =====
10760: # ANÁLISIS CONTRAFACTUAL (Pearl's Three-Layer Causal Hierarchy)
10761: # =====
10762:
10763: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_counterfactuals")
10764: def generate_counterfactuals(self, dag: CausalDAG, causal_effects: list[CausalEffect],
10765:                             financial_analysis: dict[str, Any]) -> list[CounterfactualScenario]:
10766: """
10767: Genera escenarios contrafactuals usando el framework de Pearl (2009)
10768: Level 3 - Counterfactual: "What if we had done X instead of Y?"
10769:
10770: Implementación basada en:
10771: - Pearl & Mackenzie (2018) - The Book of Why
10772: - Sharma & Kiciman (2020) - DoWhy: An End-to-End Library for Causal Inference
10773: """
10774: print("δ\237\224 Generando escenarios contrafactuals...")
10775:
10776: scenarios = []
10777: G = dag.graph
10778: pillar_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'pilar']
10779:
10780: current_budgets = {
10781:     node: float(dag.nodes[node].associated_budget) if dag.nodes[node].associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_
10782: viabilidad_tablas.PDET MunicipalPlanAnalyzer.generate_counterfactuals", "auto_param_L1438_101", 0.0)
10783:     for node in pillar_nodes
10784: }
10785: total_budget = sum(current_budgets.values())
10786:
10787: if total_budget == 0:
10788:     print("⚠️ No hay información presupuestal para contrafactuals")
10789:     return scenarios
10790:
10791: # Escenario 1: Incremento proporcional del 20%
intervention_1 = {node: budget * 1.2 for node, budget in current_budgets.items()}

```

```

10792:     scenario_1 = self._simulate_intervention(intervention_1, dag, causal_effects, "Incremento 20% presupuesto")
10793:     scenarios.append(scenario_1)
10794:
10795:     # Escenario 2: Rebalanceo hacia educaciÃ³n y salud
10796:     priority_pillars = ['EducaciÃ³n rural y primera infancia', 'Salud rural']
10797:     intervention_2 = current_budgets.copy()
10798:     for pillar in priority_pillars:
10799:         if pillar in intervention_2:
10800:             intervention_2[pillar] *= 1.5
10801:
10802:     other_reduction = (sum(intervention_2.values()) - total_budget) / max(
10803:         len(intervention_2) - len(priority_pillars), 1)
10804:     for pillar in intervention_2:
10805:         if pillar not in priority_pillars:
10806:             intervention_2[pillar] = max(intervention_2[pillar] - other_reduction, 0)
10807:
10808:     scenario_2 = self._simulate_intervention(intervention_2, dag, causal_effects,
10809:                                              "PriorizaciÃ³n educaciÃ³n y salud")
10810:     scenarios.append(scenario_2)
10811:
10812:     # Escenario 3: FocalizaciÃ³n en pilar de mayor impacto
10813:     if causal_effects:
10814:         best_effect = max(causal_effects, key=lambda e: e.probability_positive * abs(e.posterior_mean))
10815:         best_pillar = best_effect.treatment
10816:
10817:         intervention_3 = {node: budget * ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer.generate_counterfactuals", "auto_param_L1474_45", 0.7) for node, budget in current_budgets.items()}
10818:         if best_pillar in intervention_3:
10819:             intervention_3[best_pillar] = current_budgets[best_pillar] * 1.8
10820:
10821:         scenario_3 = self._simulate_intervention(intervention_3, dag, causal_effects,
10822:                                              f"FocalizaciÃ³n en {best_pillar[:40]}")
10823:         scenarios.append(scenario_3)
10824:
10825:     print(f"\u202a{len(scenarios)} escenarios contrafactuals generados")
10826:     return scenarios
10827:
10828: @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._simulate_intervention")
10829: def _simulate_intervention(self, intervention: dict[str, float], dag: CausalDAG,
10830:                           causal_effects: list[CausalEffect], description: str) -> CounterfactualScenario:
10831: """
10832: Simula intervenciÃ³n usando do-calculus (Pearl, 2009)
10833: Implementa: P(Y | do(X=x)) mediante propagaciÃ³n por el DAG
10834: """
10835: G = dag.graph
10836: predicted_outcomes = {}
10837:
10838: outcome_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'outcome']
10839:
10840: for outcome in outcome_nodes:
10841:     relevant_effects = [e for e in causal_effects if e.outcome == outcome]
10842:
10843:     if not relevant_effects:
10844:         continue
10845:
10846:     expected_change = ParameterLoaderV2.get("farfan_core.analysis.financiero_viabilidad_tablas.PDET MunicipalPlanAnalyzer._simulate_intervention", "e

```

```
xpected_change", 0.0) # Refactored
10847:         variance_sum = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._simulate_intervention", "vari
ance_sum", 0.0) # Refactored
10848:
10849:         for effect in relevant_effects:
10850:             treatment = effect.treatment
10851:             if treatment not in intervention:
10852:                 continue
10853:
10854:             current_budget = float(dag.nodes[treatment].associated_budget) if dag.nodes[
10855:                 treatment].associated_budget else ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._si
mulate_intervention", "auto_param_L1512_54", 0.0)
10856:             new_budget = intervention[treatment]
10857:
10858:             budget_multiplier = new_budget / current_budget if current_budget > 0 else ParameterLoaderV2.get("farfan_core.analysis.financiero_viability
_tablas.PDET MunicipalPlanAnalyzer._simulate_intervention", "auto_param_L1515_91", 1.0)
10859:
10860:             # Rendimientos decrecientes: log transform
10861:             effect_multiplier = np.log1p(budget_multiplier) / np.log1p(ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Muni
cipalPlanAnalyzer._simulate_intervention", "auto_param_L1518_75", 1.0))
10862:
10863:             expected_change += effect.posterior_mean * effect_multiplier
10864:
10865:             ci_width = effect.credible_interval_95[1] - effect.credible_interval_95[0]
10866:             variance_sum += (ci_width / 3.92) ** 2 # 95% CI \sqrt{110} 3.92 std
10867:
10868:             predicted_std = np.sqrt(variance_sum)
10869:             predicted_outcomes[outcome] = (
10870:                 expected_change,
10871:                 expected_change - 1.96 * predicted_std,
10872:                 expected_change + 1.96 * predicted_std
10873:             )
10874:
10875:             probability_improvement = {}
10876:             for outcome, (mean, lower, upper) in predicted_outcomes.items():
10877:                 scale = (upper - lower) / 3.92
10878:                 if scale <= 0: scale = 1e-9
10879:                 prob_positive = stats.norm.sf(0, loc=mean, scale=scale)
10880:                 probability_improvement[outcome] = float(prob_positive)
10881:
10882:             narrative = self._generate_scenario_narrative(description, intervention, predicted_outcomes,
10883:                                               probability_improvement)
10884:
10885:             return CounterfactualScenario(
10886:                 intervention=intervention,
10887:                 predicted_outcomes=predicted_outcomes,
10888:                 probability_improvement=probability_improvement,
10889:                 narrative=narrative
10890:             )
10891:
10892: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._generate_scenario_narrative")
10893: def _generate_scenario_narrative(self, description: str, intervention: dict[str, float],
10894:                                     predicted_outcomes: dict[str, tuple[float, float, float]],
10895:                                     probabilities: dict[str, float]) -> str:
10896:     """Genera narrativa interpretable del escenario contrafactual"""
10897:
```

```

10898:     narrative = f"**{description}**\n\n"
10899:     narrative += "***IntervenciÃ³n propuesta:**\n"
10900:
10901:     total_intervention = sum(intervention.values())
10902:     for pillar, budget in sorted(intervention.items(), key=lambda x: -x[1])[:5]:
10903:         percentage = (budget / total_intervention * 100) if total_intervention > 0 else 0
10904:         narrative += f"- {pillar[:50]}: ${budget:.0f} COP ({percentage:.1f}%) \n"
10905:
10906:     narrative += "\n**Efectos esperados:**\n"
10907:
10908:     significant_outcomes = [(o, p) for o, p in probabilities.items() if p > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDE"
10909: TMunicipalPlanAnalyzer._generate_scenario_narrative", "auto_param_L1565_80", 0.6)]
10910:     significant_outcomes.sort(key=lambda x: -x[1])
10911:
10912:     for outcome, prob in significant_outcomes[:5]:
10913:         mean, lower, upper = predicted_outcomes[outcome]
10914:         narrative += f"- {outcome}: {mean:+.2f} (IC95%: [{lower:.2f}, {upper:.2f}]) - "
10915:         narrative += f"\nProbabilidad de mejora: {prob * 100:.0f}%\n"
10916:
10917:     return narrative
10918: # =====
10919: # ANÃLISIS DE SENSIBILIDAD (Cinelli et al., 2022)
10920: # =====
10921:
10922: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.sensitivity_analysis")
10923: def sensitivity_analysis(self, causal_effects: list[CausalEffect], dag: CausalDAG) -> dict[str, Any]:
10924:
10925:     """
10926:     AnÃ¡lisis de sensibilidad para supuestos de identificaciÃ³n causal
10927:     Basado en: Cinelli, Forney & Pearl (2022) - "A Crash Course in Good and Bad Controls"
10928:
10929:     print("Executing sensitivity analysis...")
10930:
10931:     sensitivity_results = {}
10932:
10933:     for effect in causal_effects[:10]: # Top 10 effects
10934:         unobserved_confounding = self._compute_e_value(effect)
10935:
10936:         robustness_value = self._compute_robustness_value(effect, dag)
10937:
10938:         sensitivity_results[f"{effect.treatment[:30]}\206\222{effect.outcome[:30]}"] = {
10939:             'e_value': unobserved_confounding,
10940:             'robustness_value': robustness_value,
10941:             'interpretation': self._interpret_sensitivity(unobserved_confounding, robustness_value)
10942:         }
10943:
10944:     print(f"\234\223 Sensibilidad analizada para {len(sensitivity_results)} efectos")
10945:
10946: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_e_value")
10947: def _compute_e_value(self, effect: CausalEffect) -> float:
10948:
10949:     """
10950:     E-value: mÃ¬nima fuerza de confounding no observado para anular el efecto
10951:     FÃ³rmula: E = effect_estimate + sqrt(effect_estimate * (effect_estimate - 1))
10952:
10953:     Referencia: VanderWeele & Ding (2017) - Ann Intern Med

```

```

10953:     """
10954:     if effect.posterior_mean <= 0:
10955:         return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_e_value", "auto_param_L1612_1
9", 1.0)
10956:
10957:     rr = np.exp(effect.posterior_mean) # Convert log-scale to risk ratio
10958:     if rr <= 1:
10959:         return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_e_value", "auto_param_L1616_1
9", 1.0)
10960:     e_value = rr + np.sqrt(rr * (rr - 1))
10961:
10962:     return float(e_value)
10963:
10964: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_robustness_value")
10965: def _compute_robustness_value(self, effect: CausalEffect, dag: CausalDAG) -> float:
10966:     """
10967:     Robustness Value: percentil de la distribuciÃ³n posterior que cruza cero
10968:     Valores altos (>ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_robustness_value", "auto
_param_L1625_24", 0.95)) indican alta robustez
10969:     """
10970:     ci_lower, ci_upper = effect.credible_interval_95
10971:
10972:     if ci_lower > 0 or ci_upper < 0:
10973:         return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_robustness_value", "auto_para
m_L1630_19", 1.0)
10974:
10975:     width = ci_upper - ci_lower
10976:     if width == 0:
10977:         return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_robustness_value", "auto_para
m_L1634_19", 0.5)
10978:
10979:     robustness = abs(effect.posterior_mean) / (width / 2)
10980:     return float(min(robustness, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._compute_robustness_
value", "auto_param_L1637_37", 1.0)))
10981:
10982: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_sensitivity")
10983: def _interpret_sensitivity(self, e_value: float, robustness: float) -> str:
10984:     """InterpretaciÃ³n de resultados de sensibilidad"""
10985:
10986:     if e_value > 2.0 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_sens
itivit", "auto_param_L1643_42", 0.8):
10987:         return "Efecto robusto - Resistente a confounding no observado"
10988:     elif e_value > 1.5 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_se
nsitivity", "auto_param_L1645_44", 0.6):
10989:         return "Efecto moderadamente robusto - PrecauciÃ³n con confounders"
10990:     elif e_value > 1.2 and robustness > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._interpret_se
nsitivity", "auto_param_L1647_44", 0.4):
10991:         return "Efecto sensible - Alta vulnerabilidad a confounding"
10992:     else:
10993:         return "Efecto frÃ¡gil - Resultados no confiables sin ajustes adicionales"
10994:
10995: # =====
10996: # SCORING INTEGRAL DE CALIDAD
10997: # =====
10998:
10999: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.calculate_quality_score")

```

```
11000:     def calculate_quality_score(self, text: str, tables: list[ExtractedTable],
11001:                                     financial_analysis: dict[str, Any],
11002:                                     responsible_entities: list[ResponsibleEntity],
11003:                                     causal_dag: CausalDAG,
11004:                                     causal_effects: list[CausalEffect]) -> QualityScore:
11005:         """
11006:             Puntaje bayesiano integral de calidad del PDM
11007:             Integra todas las dimensiones de análisis con pesos calibrados
11008:         """
11009:         print("â\220 Calculando score integral de calidad...")
11010:
11011:         financial_score = self._score_financial_component(financial_analysis)
11012:         indicator_score = self._score_indicators(tables, text)
11013:         responsibility_score = self._score_responsibility_clarity(responsible_entities)
11014:         temporal_score = self._score_temporal_consistency(text, tables)
11015:         pdet_score = self._score_pdet_alignment(text, tables, causal_dag)
11016:         causal_score = self._score_causal_coherence(causal_dag, causal_effects)
11017:
11018:         weights = np.array([ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_28", 0.20), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_34", 0.15), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_40", 0.15), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_46", 0.10), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_52", 0.20), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer.calculate_quality_score", "auto_param_L1675_58", 0.20)])
11019:         scores = np.array([
11020:             financial_score, indicator_score, responsibility_score,
11021:             temporal_score, pdet_score, causal_score
11022:         ])
11023:
11024:         overall_score = float(np.dot(weights, scores))
11025:
11026:         confidence = self._estimate_score_confidence(scores, weights)
11027:
11028:         evidence = {
11029:             'financial': financial_score,
11030:             'indicators': indicator_score,
11031:             'responsibility': responsibility_score,
11032:             'temporal': temporal_score,
11033:             'pdet_alignment': pdet_score,
11034:             'causal_coherence': causal_score
11035:         }
11036:
11037:         print(f" â\234\223 Score final: {overall_score:.2f}/10.0")
11038:
11039:         return QualityScore(
11040:             overall_score=overall_score,
11041:             financial_feasibility=financial_score,
11042:             indicator_quality=indicator_score,
11043:             responsibility_clarity=responsibility_score,
11044:             temporal_consistency=temporal_score,
11045:             pdet_alignment=pdet_score,
11046:             causal_coherence=causal_score,
11047:             confidence_interval=confidence,
11048:             evidence=evidence
11049:         )
11050:
```

```

11051:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_financial_component")
11052:     def _score_financial_component(self, financial_analysis: dict[str, Any]) -> float:
11053:         """Score componente financiero (0-10)"""
11054:
11055:         budget = financial_analysis.get('total_budget', 0)
11056:         if budget == 0:
11057:             return ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_financial_component", "auto_param_L1714_19", 0.0)
11058:
11059:         budget_score = min(np.log10(float(budget)) / 12, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_financial_component", "auto_param_L1716_57", 1.0)) * 3.0
11060:
11061:         diversity = financial_analysis['funding_sources'].get('diversity_index', 0)
11062:         max_diversity = financial_analysis['funding_sources'].get('max_diversity', 1)
11063:         diversity_score = (diversity / max(max_diversity, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_financial_component", "auto_param_L1720_58", 0.1))) * 3.0
11064:
11065:         sustainability = financial_analysis.get('sustainability_score', 0)
11066:         sustainability_score = sustainability * 2.5
11067:
11068:         risk = financial_analysis['risk_assessment'].get('risk_score', ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_financial_component", "auto_param_L1725_71", 0.5))
11069:         risk_score = (1 - risk) * 1.5
11070:
11071:         return float(min(budget_score + diversity_score + sustainability_score + risk_score, 1.0))
11072:
11073:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_indicators")
11074:     def _score_indicators(self, tables: list[ExtractedTable], text: str) -> float:
11075:         """Score calidad de indicadores (0-10)"""
11076:
11077:         indicator_tables = [t for t in tables if t.table_type == 'indicadores']
11078:
11079:         if not indicator_tables:
11080:             baseline_mentions = len(re.findall(r'l[Á-i]nea\s+base', text, re.IGNORECASE))
11081:             meta_mentions = len(re.findall(r'meta', text, re.IGNORECASE))
11082:
11083:             if baseline_mentions > 5 and meta_mentions > 5:
11084:                 return 4.0
11085:             return 2.0
11086:
11087:             completeness_score = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_indicators", "completeness_score", 0.0) # Refactored
11088:             for table in indicator_tables:
11089:                 df = table.df
11090:                 required_cols = ['indicador', 'lÃ-nea base', 'meta', 'fuente']
11091:                 present_cols = sum(1 for col in required_cols if any(col in str(c).lower() for c in df.columns))
11092:                 completeness_score += (present_cols / len(required_cols)) * 3.0
11093:
11094:             completeness_score = min(completeness_score, 4.0)
11095:
11096:             smart_patterns = [
11097:                 r'\d+%', # Percentages
11098:                 r'\d+\s+(?:personas|hogares|familias|hectÃreas)', # Quantities
11099:                 r'reducir|aumentar|mejorar|incrementar', # Action verbs
11100:             ]
11101:

```

```

11102:     smart_count = sum(len(re.findall(pattern, text, re.IGNORECASE)) for pattern in smart_patterns)
11103:     smart_score = min(smart_count / 50, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_indicators", "auto_param_L1760_44", 1.0)) * 3.0
11104:
11105:     formula_mentions = len(re.findall(r'f[Ã³]rmula', text, re.IGNORECASE))
11106:     periodicity_mentions = len(re.findall(r'periodicidad|trimestral|anual|mensual', text, re.IGNORECASE))
11107:
11108:     technical_score = min((formula_mentions + periodicity_mentions) / 10, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_indicators", "auto_param_L1765_78", 1.0)) * 3.0
11109:
11110:     return float(min(completeness_score + smart_score + technical_score, 10.0))
11111:
11112: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_responsibility_clarity")
11113: def _score_responsibility_clarity(self, entities: list[ResponsibleEntity]) -> float:
11114:     """Score claridad de responsables (0-10)"""
11115:
11116:     if not entities:
11117:         return 2.0
11118:
11119:     count_score = min(len(entities) / 15, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_responsibility_clarity", "auto_param_L1776_46", 1.0)) * 3.0
11120:
11121:     avg_specificity = np.mean([e.specificity_score for e in entities])
11122:     specificity_score = avg_specificity * 4.0
11123:
11124:     institutional_entities = [e for e in entities if e.entity_type in ['secretarÃ-a', 'direcciÃ³n', 'oficina']]
11125:     institutional_ratio = len(institutional_entities) / max(len(entities), 1)
11126:     institutional_score = institutional_ratio * 3.0
11127:
11128:     return float(min(count_score + specificity_score + institutional_score, 10.0))
11129:
11130: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_temporal_consistency")
11131: def _score_temporal_consistency(self, text: str, tables: list[ExtractedTable]) -> float:
11132:     """Score consistencia temporal (0-10)"""
11133:
11134:     years_mentioned = set(re.findall(r'20[2-3]\d', text))
11135:
11136:     if len(years_mentioned) < 2:
11137:         return 3.0
11138:
11139:     years = [int(y) for y in years_mentioned]
11140:     year_range = max(years) - min(years) if years else 0
11141:     range_score = min(year_range / 4, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_temporal_consistency", "auto_param_L1798_42", 1.0)) * 3.0
11142:
11143:     cronograma_tables = [t for t in tables if t.table_type == 'cronograma']
11144:     cronograma_score = min(len(cronograma_tables) * 2, 4.0)
11145:
11146:     temporal_terms = ['cronograma', 'aÃ±o', 'trimestre', 'mes', 'periodo', 'etapa', 'fase']
11147:     term_count = sum(len(re.findall(rf'\b{term}\b', text, re.IGNORECASE)) for term in temporal_terms)
11148:     term_score = min(term_count / 30, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_temporal_consistency", "auto_param_L1805_42", 1.0)) * 3.0
11149:
11150:     return float(min(range_score + cronograma_score + term_score, 10.0))
11151:
11152: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._score_pdet_alignment")

```

```

11153:     def _score_pdet_alignment(self, text: str, tables: list[ExtractedTable], dag: CausalDAG) -> float:
11154:         """Score alineaciÃ³n con pilares PDET (0-10)"""
11155:
11156:         text_lower = text.lower()
11157:
11158:         pillar_mentions = {}
11159:         for pillar in self.context.PDET_PILLARS:
11160:             pillar_lower = pillar.lower()
11161:             keywords = pillar_lower.split()[:3]
11162:
11163:             count = sum(text_lower.count(kw) for kw in keywords)
11164:             pillar_mentions[pillar] = count
11165:
11166:             coverage = sum(1 for count in pillar_mentions.values() if count > 0)
11167:             coverage_score = (coverage / len(self.context.PDET_PILLARS)) * 4.0
11168:
11169:             pdet_explicit = len(re.findall(r'\bPDET\b', text, re.IGNORECASE))
11170:             patr_mentions = len(re.findall(r'\bPATR\b', text, re.IGNORECASE))
11171:             explicit_score = min((pdet_explicit + patr_mentions) / 15, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._score_pdet_alignment", "auto_param_L1828_67", 1.0)) * 3.0
11172:
11173:             pdet_tables = [t for t in tables if t.table_type == 'pdet']
11174:             table_score = min(len(pdet_tables) * 1.5, 3.0)
11175:
11176:             return float(min(coverage_score + explicit_score + table_score, 10.0))
11177:
11178:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence")
11179:     def _score_causal_coherence(self, dag: CausalDAG, effects: list[CausalEffect]) -> float:
11180:         """Score coherencia causal del plan (0-10)"""
11181:
11182:         G = dag.graph
11183:
11184:         if G.number_of_nodes() == 0:
11185:             return 2.0
11186:
11187:         structure_score = min(G.number_of_edges() / (G.number_of_nodes() * 1.5), ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence", "auto_param_L1844_81", 1.0)) * 3.0
11188:
11189:         if not effects:
11190:             effect_quality = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._score_causal_coherence", "effect_quality", 0.0) # Refactored
11191:         else:
11192:             avg_probability = np.mean([e.probability_significant for e in effects])
11193:             effect_quality = avg_probability * 4.0
11194:
11195:         pillar_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'pillar']
11196:         outcome_nodes = [n for n, data in G.nodes(data=True) if data.get('type') == 'outcome']
11197:
11198:         connected_pillars = sum(1 for p in pillar_nodes if any(nx.has_path(G, p, o) for o in outcome_nodes))
11199:         connectivity = (connected_pillars / max(len(pillar_nodes), 1)) * 3.0
11200:
11201:         return float(min(structure_score + effect_quality + connectivity, 10.0))
11202:
11203:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDETMunicipalPlanAnalyzer._estimate_score_confidence")
11204:     def _estimate_score_confidence(self, scores: np.ndarray, weights: np.ndarray) -> tuple[float, float]:
11205:         """Estima intervalo de confianza para el score usando bootstrap"""

```

```

11206:
11207:     n_bootstrap = 1000
11208:     bootstrap_scores = []
11209:
11210:     for _ in range(n_bootstrap):
11211:         noise = np.random.normal(0, ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._estimate_score_c
onfidence", "auto_param_L1868_40", 0.5), size=len(scores))
11212:         noisy_scores = np.clip(scores + noise, 0, 10)
11213:
11214:         bootstrap_score = np.dot(weights, noisy_scores)
11215:         bootstrap_scores.append(bootstrap_score)
11216:
11217:     ci_lower, ci_upper = np.percentile(bootstrap_scores, [2.5, 97.5])
11218:
11219:     return (float(ci_lower), float(ci_upper))
11220:
11221: # =====
11222: # EXPORTACIÃ“N Y VISUALIZACIÃ“N
11223: # =====
11224:
11225: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.export_causal_network")
11226: def export_causal_network(self, dag: CausalDAG, output_path: str) -> None:
11227:     """Exporta el DAG causal en formato GraphML para Gephi/Cytoscape"""
11228:
11229:     G = dag.graph.copy()
11230:
11231:     for node, data in G.nodes(data=True):
11232:         data['label'] = node[:50]
11233:         data['node_type'] = data.get('type', 'unknown')
11234:         data['budget'] = data.get('budget', ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.export_ca
usal_network", "auto_param_L1891_48", 0.0))
11235:
11236:         for _u, _v, data in G.edges(data=True):
11237:             data['weight'] = data.get('probability', ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.expo
rt_causal_network", "auto_param_L1894_53", 0.5))
11238:             data['edge_type'] = data.get('type', 'unknown')
11239:
11240:             nx.write_graphml(G, output_path)
11241:             print(f"â\234\205 Red causal exportada a: {output_path}")
11242:
11243: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_executive_report")
11244: def generate_executive_report(self, analysis_results: dict[str, Any]) -> str:
11245:     """Genera reporte ejecutivo en Markdown"""
11246:
11247:     report = "# ANÃ“20LISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET\n\n"
11248:     report += f"**Fecha de anÃ¡lisis:** {datetime.now().strftime('%Y-%m-%d %H:%M')}\n\n"
11249:
11250:     report += "## 1. RESUMEN EJECUTIVO\n\n"
11251:
11252:     quality = analysis_results['quality_score']
11253:     report += f"**Score Global de Calidad:** {quality['overall_score']:.2f}/10.0 "
11254:     report += f"(IC95%: [{quality['confidence_interval'][0]:.2f}, {quality['confidence_interval'][1]:.2f}])\n\n"
11255:
11256:     report += self._interpret_overall_quality(quality['overall_score'])
11257:     report += "\n\n"
11258:
```

```

11259:     report += "### Dimensiones Evaluadas\n\n"
11260:     report += f"- **Viabilidad Financiera:** {quality['financial_feasibility']:.1f}/10\n"
11261:     report += f"- **Calidad de Indicadores:** {quality['indicator_quality']:.1f}/10\n"
11262:     report += f"- **Claridad de Responsables:** {quality['responsibility_clarity']:.1f}/10\n"
11263:     report += f"- **Consistencia Temporal:** {quality['temporal_consistency']:.1f}/10\n"
11264:     report += f"- **AlineaciÃ³n PDET:** {quality['pdet_alignment']:.1f}/10\n"
11265:     report += f"- **Coherencia Causal:** {quality['causal_coherence']:.1f}/10\n\n"
11266:
11267:     report += "## 2. ANÃLISIS FINANCIERO\n\n"
11268:     fin = analysis_results['financial_analysis']
11269:     report += f"**Presupuesto Total:** ${fin['total_budget']:.0f} COP\n\n"
11270:
11271:     report += "## DistribuciÃ³n por Fuente\n\n"
11272:     if fin['funding_sources'] and fin['funding_sources']['distribution']:
11273:         for source, amount in sorted(fin['funding_sources']['distribution'].items(), key=lambda x: -x[1])[:5]:
11274:             pct = (amount / fin['total_budget']) * 100 if fin['total_budget'] > 0 else 0
11275:             report += f"- {source}: ${amount:.0f} ({pct:.1f}%) \n"
11276:
11277:     report += f"\n**Ãndice de DiversificaciÃ³n:** {fin['funding_sources'].get('diversity_index', 0):.2f}\n"
11278:     report += f"**Score de Sostenibilidad:** {fin['sustainability_score']:.2f}\n"
11279:     report += f"**EvaluaciÃ³n de Riesgo:** {fin['risk_assessment']['interpretation']}\n\n"
11280:
11281:     report += "## 3. INFERENCIA CAUSAL\n\n"
11282:
11283:     effects = analysis_results.get('causal_effects', [])
11284:     if effects:
11285:         report += "## Efectos Causales Principales\n\n"
11286:
11287:     significant_effects = [e for e in effects if e['probability_significant'] > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_ta
blas.PDET MunicipalPlanAnalyzer.generate_executive_report", "auto_param_L1944_88", 0.7)]
11288:     significant_effects.sort(key=lambda e: abs(e['posterior_mean']), reverse=True)
11289:
11290:     for effect in significant_effects[:5]:
11291:         report += f"**{effect['treatment'][:40]} a 2022 {effect['outcome'][:40]}**\n"
11292:         report += f"- Efecto estimado: {effect['posterior_mean']:+.3f} "
11293:         report += f"(IC95%: [{effect['credible_interval'][0]:.3f}, {effect['credible_interval'][1]:.3f}])\n"
11294:         report += f"- Probabilidad de efecto positivo: {effect['probability_positive']} * 100:.0f%\n"
11295:
11296:         if effect['mediating_paths']:
11297:             report += f"- VÃ¡-as de mediaciÃ³n: {len(effect['mediating_paths'])} \n"
11298:             report += "\n"
11299:
11300:     report += "## 4. ESCENARIOS CONTRAFACTUALES\n\n"
11301:
11302:     scenarios = analysis_results.get('counterfactuals', [])
11303:     for _i, scenario in enumerate(scenarios, 1):
11304:         report += scenario['narrative']
11305:         report += "\n--\n"
11306:
11307:     report += "## 5. ANÃLISIS DE SENSIBILIDAD\n\n"
11308:
11309:     sensitivity = analysis_results.get('sensitivity_analysis', {})
11310:     if sensitivity:
11311:         report += "| RelaciÃ³n Causal | E-Value | Robustez | InterpretaciÃ³n |\n"
11312:         report += "-----|-----|-----|-----|\n"
11313:

```

```

11314:         for relation, metrics in list(sensitivity.items())[:8]:
11315:             report += f"| {relation} | {metrics['e_value']:.2f} | {metrics['robustness_value']:.2f} | {metrics['interpretation'][:50]} | \n"
11316:
11317:             report += "\n## 6. RECOMENDACIONES\n\n"
11318:             report += self._generate_recommendations(analysis_results)
11319:
11320:             report += "\n---\n"
11321:             report += "*Análisis generado por PDET Municipal Plan Analyzer v5.0*\n"
11322:             report += "*Metodología: Inferencia Causal Bayesiana + Structural Causal Models*\n"
11323:
11324:             return report
11325:
11326: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._interpret_overall_quality")
11327: def _interpret_overall_quality(self, score: float) -> str:
11328:     """Interpretación del score global"""
11329:
11330:     if score >= 8.0:
11331:         return ("**Evaluación: EXCELENTE** \u2192 205\n\n"
11332:                 "El plan cumple con altos estándares de calidad técnica. "
11333:                 "Presenta coherencia causal sólida, viabilidad financiera demostrable, "
11334:                 "y alineación robusta con los pilares PDET.")
11335:     elif score >= 6.5:
11336:         return ("**Evaluación: BUENO** \u2192 223\n\n"
11337:                 "El plan presenta bases sólidas pero con oportunidades de mejora. "
11338:                 "Se recomienda fortalecer algunos componentes específicos.")
11339:     elif score >= 5.0:
11340:         return ("**Evaluación: ACEPTABLE** \u2192 217\n\n"
11341:                 "El plan cumple requisitos mínimos pero requiere ajustes sustanciales "
11342:                 "en múltiples dimensiones para asegurar efectividad.")
11343:     else:
11344:         return ("**Evaluación: DEFICIENTE** \u2192 214\n\n"
11345:                 "El plan presenta deficiencias críticas que comprometen su viabilidad. "
11346:                 "Se requiere reformulación integral.")
11347:
11348: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._generate_recommendations")
11349: def _generate_recommendations(self, analysis_results: dict[str, Any]) -> str:
11350:     """Genera recomendaciones específicas basadas en el análisis"""
11351:
11352:     recommendations = []
11353:     quality = analysis_results['quality_score']
11354:
11355:     # Recomendaciones financieras
11356:     if quality['financial_feasibility'] < 6.0:
11357:         fin = analysis_results['financial_analysis']
11358:         if fin['funding_sources'].get('dependency_risk', 0) > ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._generate_recommendations", "auto_param_L2015_66", 0.6):
11359:             recommendations.append(
11360:                 "**Diversificación de fuentes:** Reducir dependencia excesiva de fuentes Áreas. "
11361:                 "Explorar alternativas como cooperación internacional, APP, o gestión de recursos propios."
11362:             )
11363:
11364:             if fin['sustainability_score'] < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer._generate_recommendations", "auto_param_L2021_45", 0.5):
11365:                 recommendations.append(
11366:                     "**Sostenibilidad fiscal:** Fortalecer componente de recursos propios. "
11367:                     "Desarrollar estrategias de generación de ingresos municipales."

```

```

11368:         )
11369:
11370:     # Recomendaciones de indicadores
11371:     if quality['indicator_quality'] < 6.0:
11372:         recommendations.append(
11373:             """Fortalecimiento de indicadores:** Definir indicadores SMART completos "
11374:             "(especÃ-ficos, medibles, alcanzables, relevantes, temporales) con lÃ-neas base, "
11375:             "metas cuantificadas, fÃ³rmulas de cÃ¡lculo y fuentes verificables."
11376:         )
11377:
11378:     # Recomendaciones causales
11379:     effects = analysis_results.get("causal_effects", [])
11380:     if effects:
11381:         weak_effects = [e for e in effects if e['probability_significant'] < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PD
ETMunicipalPlanAnalyzer._generate_recommendations", "auto_param_L2038_81", 0.5)]
11382:
11383:         if len(weak_effects) > len(effects) * ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._genera
te_recommendations", "auto_param_L2040_50", 0.5):
11384:             recommendations.append(
11385:                 """Robustez causal:** Fortalecer vÃ-nculos entre intervenciones y resultados esperados. "
11386:                 "Explicitar teorÃ-as de cambio y mecanismos causales subyacentes."
11387:             )
11388:
11389:     # Recomendaciones PDET
11390:     if quality['pdet_alignment'] < 6.0:
11391:         recommendations.append(
11392:             """AlineaciÃ³n PDET:** Articular explÃ-citamente con los 8 pilares del Pacto Estructurante. "
11393:             "Referenciar iniciativas PATR y asegurar coherencia con transformaciÃ³n territorial."
11394:         )
11395:
11396:     # Recomendaciones de responsabilidad
11397:     if quality['responsibility_clarity'] < 6.0:
11398:         recommendations.append(
11399:             """Claridad institucional:** Especificar responsables concretos para cada programa. "
11400:             "Evitar asignaciones genÃ©ricas como 'todas las secretarÃ-as' o 'alcaldÃ-a municipal'."
11401:         )
11402:
11403:     # Recomendaciones de mejores escenarios
11404:     scenarios = analysis_results.get('counterfactuals', [])
11405:     if scenarios:
11406:         best_scenario = max(scenarios,
11407:                               key=lambda s: sum(s['probability_improvement'].values()))
11408:
11409:         recommendations.append(
11410:             f"""OptimizaciÃ³n presupuestal:** Considerar escenario '{best_scenario['narrative'].split('**')[1]}'
11411:             "que maximiza probabilidad de impacto en outcomes clave."
11412:         )
11413:
11414:     if not recommendations:
11415:         return "El plan presenta solidez en todas las dimensiones evaluadas. Continuar con implementaciÃ³n segÃºn lo planificado.\n"
11416:
11417:     result = ""
11418:     for i, rec in enumerate(recommendations, 1):
11419:         result += f"{i}. {rec}\n\n"
11420:
11421:     return result

```

```
11422:  
11423:     # =====  
11424:     # PIPELINE PRINCIPAL  
11425:     # =====  
11426:  
11427:     @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET Municipal Plan Analyzer.analyze_municipal_plan_sync")  
11428:     def analyze_municipal_plan_sync(self, pdf_path: str, output_dir: str | None = None) -> dict[str, Any]:  
11429:         """Synchronous wrapper for analyze_municipal_plan."""  
11430:  
11431:             loop = asyncio.new_event_loop()  
11432:             try:  
11433:                 return loop.run_until_complete(self.analyze_municipal_plan(pdf_path, output_dir))  
11434:             finally:  
11435:                 loop.close()  
11436:  
11437:     async def analyze_municipal_plan(self, pdf_path: str, output_dir: str | None = None) -> dict[str, Any]:  
11438:         """  
11439:             Pipeline completo de anÃ¡lisis  
11440:  
11441:             Args:  
11442:                 pdf_path: Ruta al PDF del Plan de Desarrollo Municipal  
11443:                 output_dir: Directorio para guardar outputs (opcional)  
11444:  
11445:             Returns:  
11446:                 Diccionario con todos los resultados del anÃ¡lisis  
11447:         """  
11448:  
11449:         print("\n" + "=" * 70)  
11450:         print("ANÃLISIS INTEGRAL - PLAN DE DESARROLLO MUNICIPAL PDET")  
11451:         print("=" * 70 + "\n")  
11452:  
11453:         start_time = datetime.now()  
11454:  
11455:         # 1. ExtracciÃ³n de texto  
11456:         print("δ\237\223\204 Extrayendo texto del PDF...")  
11457:         full_text = self._extract_full_text(pdf_path)  
11458:         print(f" {len(full_text)} caracteres extraÃ±os\n")  
11459:  
11460:         # 2. ExtracciÃ³n de tablas  
11461:         tables = await self.extract_tables(pdf_path)  
11462:  
11463:         # 3. AnÃ¡lisis financiero  
11464:         financial_analysis = self.analyze_financial_feasibility(tables, full_text)  
11465:  
11466:         # 4. IdentificaciÃ³n de responsables  
11467:         responsible_entities = self.identify_responsible_entities(full_text, tables)  
11468:  
11469:         # 5. ConstrucciÃ³n de DAG causal  
11470:         causal_dag = self.construct_causal_dag(full_text, tables, financial_analysis)  
11471:  
11472:         # 6. EstimaciÃ³n de efectos causales  
11473:         causal_effects = self.estimate_causal_effects(causal_dag, full_text, financial_analysis)  
11474:  
11475:         # 7. GeneraciÃ³n de contrafactuals  
11476:         counterfactuals = self.generate_counterfactuals(causal_dag, causal_effects, financial_analysis)  
11477:
```

```

11478:     # 8. Análisis de sensibilidad
11479:     sensitivity_analysis = self.sensitivity_analysis(causal_effects, causal_dag)
11480:
11481:     # 9. Score integral de calidad
11482:     quality_score = self.calculate_quality_score(
11483:         full_text, tables, financial_analysis, responsible_entities,
11484:         causal_dag, causal_effects
11485:     )
11486:
11487:     # 10. Compilación de resultados
11488:     results = {
11489:         'metadata': {
11490:             'pdf_path': pdf_path,
11491:             'analysis_date': datetime.now().isoformat(),
11492:             'processing_time_seconds': (datetime.now() - start_time).total_seconds(),
11493:             'analyzer_version': '5.0'
11494:         },
11495:         'extraction': {
11496:             'text_length': len(full_text),
11497:             'tables_extracted': len(tables),
11498:             'table_types': {t.table_type: sum(1 for x in tables if x.table_type == t.table_type)
11499:                             for t in tables if t.table_type}
11500:         },
11501:         'financial_analysis': financial_analysis,
11502:         'responsible_entities': [self._entity_to_dict(e) for e in responsible_entities[:20]],
11503:         'causal_dag': {
11504:             'nodes': len(causal_dag.nodes),
11505:             'edges': len(causal_dag.edges),
11506:             'pillar_nodes': [n for n, node in causal_dag.nodes.items() if node.node_type == 'pillar'],
11507:             'outcome_nodes': [n for n, node in causal_dag.nodes.items() if node.node_type == 'outcome']
11508:         },
11509:         'causal_effects': [self._effect_to_dict(e) for e in causal_effects[:15]],
11510:         'counterfactuals': [self._scenario_to_dict(s) for s in counterfactuals],
11511:         'sensitivity_analysis': sensitivity_analysis,
11512:         'quality_score': self._quality_to_dict(quality_score)
11513:     }
11514:
11515:     # 11. Exportación de resultados
11516:     if output_dir:
11517:         output_path = Path(output_dir)
11518:         output_path.mkdir(parents=True, exist_ok=True)
11519:
11520:         # Exportar DAG
11521:         dag_path = output_path / "causal_network.graphml"
11522:         self.export_causal_network(causal_dag, str(dag_path))
11523:
11524:         # Exportar reporte
11525:         # Delegate to factory for I/O operation
11526:         from farfan_pipeline.analysis.factory import save_json, write_text_file
11527:
11528:         report = self.generate_executive_report(results)
11529:         report_path = output_path / "executive_report.md"
11530:         write_text_file(report, report_path)
11531:         print(f"\234\205 Reporte ejecutivo guardado en: {report_path}")
11532:
11533:         # Exportar JSON

```

```

11534:         json_path = output_path / "analysis_results.json"
11535:         save_json(results, json_path)
11536:         print(f"234\205 Resultados JSON guardados en: {json_path}")
11537:
11538:         elapsed = (datetime.now() - start_time).total_seconds()
11539:         print(f"\n217 Análisis completado en {elapsed:.1f} segundos")
11540:         print("=" * 70 + "\n")
11541:
11542:     return results
11543:
11544: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._extract_full_text")
11545: def _extract_full_text(self, pdf_path: str) -> str:
11546:     """Extrae texto completo del PDF usando múltiples métodos"""
11547:
11548:     text_parts = []
11549:
11550:     # Método 1: PyMuPDF (rápido y eficiente)
11551:     # Delegar a la fábrica para operaciones I/O
11552:     from farfan_pipeline.analysis.factory import open_pdf_with_fitz, open_pdf_with_pdfplumber
11553:
11554:     try:
11555:         doc = open_pdf_with_fitz(pdf_path)
11556:         for page in doc:
11557:             text_parts.append(page.get_text())
11558:         doc.close()
11559:     except Exception as e:
11560:         print(f"232 i,217 PyMuPDF falló: {str(e)[:50]}")
11561:
11562:     # Método 2: pdfplumber (mejor para tablas complejas)
11563:     try:
11564:         pdf = open_pdf_with_pdfplumber(pdf_path)
11565:         for page in pdf.pages[:100]: # Lmite de 100 páginas
11566:             text = page.extract_text()
11567:             if text:
11568:                 text_parts.append(text)
11569:         pdf.close()
11570:     except Exception as e:
11571:         print(f"232 i,217 pdfplumber falló: {str(e)[:50]}")
11572:
11573:     full_text = '\n\n'.join(text_parts)
11574:
11575:     # Limpieza básica
11576:     full_text = re.sub(r'\n{3,}', '\n\n', full_text)
11577:     full_text = re.sub(r' {2,}', ' ', full_text)
11578:
11579:     return full_text
11580:
11581: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._entity_to_dict")
11582: def _entity_to_dict(self, entity: ResponsibleEntity) -> dict[str, Any]:
11583:     """Convierte ResponsibleEntity a diccionario"""
11584:     return {
11585:         'name': entity.name,
11586:         'type': entity.entity_type,
11587:         'specificity_score': entity.specificity_score,
11588:         'mentions': entity.mentioned_count,
11589:         'programs': entity.associated_programs,

```

```
11590:         'budget': float(entity.budget_allocated) if entity.budget_allocated else None
11591:     }
11592:
11593:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._effect_to_dict")
11594:     def _effect_to_dict(self, effect: CausalEffect) -> dict[str, Any]:
11595:         """Convierte CausalEffect a diccionario"""
11596:         return {
11597:             'treatment': effect.treatment,
11598:             'outcome': effect.outcome,
11599:             'effect_type': effect.effect_type,
11600:             'point_estimate': effect.point_estimate,
11601:             'posterior_mean': effect.posterior_mean,
11602:             'credible_interval': effect.credible_interval_95,
11603:             'probability_positive': effect.probability_positive,
11604:             'probability_significant': effect.probability_significant,
11605:             'mediating_paths': effect.mediating_paths,
11606:             'confounders_adjusted': effect.confounders_adjusted
11607:         }
11608:
11609:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._scenario_to_dict")
11610:     def _scenario_to_dict(self, scenario: CounterfactualScenario) -> dict[str, Any]:
11611:         """Convierte CounterfactualScenario a diccionario"""
11612:         return {
11613:             'intervention': scenario.intervention,
11614:             'predicted_outcomes': scenario.predicted_outcomes,
11615:             'probability_improvement': scenario.probability_improvement,
11616:             'narrative': scenario.narrative
11617:         }
11618:
11619:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._quality_to_dict")
11620:     def _quality_to_dict(self, quality: QualityScore) -> dict[str, Any]:
11621:         """Convierte QualityScore a diccionario"""
11622:         return {
11623:             'overall_score': quality.overall_score,
11624:             'financial_feasibility': quality.financial_feasibility,
11625:             'indicator_quality': quality.indicator_quality,
11626:             'responsibility_clarity': quality.responsibility_clarity,
11627:             'temporal_consistency': quality.temporal_consistency,
11628:             'pdet_alignment': quality.pdet_alignment,
11629:             'causal_coherence': quality.causal_coherence,
11630:             'confidence_interval': quality.confidence_interval,
11631:             'evidence': quality.evidence
11632:         }
11633:
11634:     @calibrated_method("farfan_core.analysis.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer._find_product_mentions")
11635:     def _find_product_mentions(self, text: str) -> list[str]:
11636:         """
11637:             Find mentions of products in text.
11638:
11639:             Args:
11640:                 text: Text to search
11641:
11642:             Returns:
11643:                 List of product mentions
11644:         """
11645:         products = []
```

```
11646:  
11647:     # Common product keywords  
11648:     product_patterns = [  
11649:         r'producto\s+(\d+)',  
11650:         r'servicio\s+(\d+)',  
11651:         r'bien\s+(\d+)',  
11652:         r'actividad\s+(\d+)',  
11653:     ]  
11654:  
11655:     for pattern in product_patterns:  
11656:         matches = re.finditer(pattern, text, re.IGNORECASE)  
11657:         for match in matches:  
11658:             products.append(match.group(0))  
11659:  
11660:     # Also look for numbered lists that might be products  
11661:     list_pattern = r'^\s*\d+\.\s+([^\n]+)'  
11662:     for match in re.finditer(list_pattern, text, re.MULTILINE):  
11663:         item_text = match.group(1).lower()  
11664:         if any(word in item_text for word in ['producto', 'servicio', 'actividad', 'bien']):  
11665:             products.append(match.group(1))  
11666:  
11667:     return products  
11668:  
11669: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer._generate_optimal_remediations")  
11670: def _generate_optimal_remediations(self, gaps: list[dict[str, Any]]) -> list[dict[str, str]]:  
11671:     """  
11672:     Generate optimal remediations for identified gaps.  
11673:  
11674:     Args:  
11675:         gaps: List of identified gaps  
11676:  
11677:     Returns:  
11678:         List of remediation recommendations  
11679:     """  
11680:     remediations = []  
11681:  
11682:     for gap in gaps:  
11683:         remediation = {  
11684:             'gap_type': gap.get('type', 'unknown'),  
11685:             'priority': 'high' if gap.get('severity') == 'high' else 'medium',  
11686:             'recommendation': ''  
11687:         }  
11688:  
11689:         gap_type = gap.get('type', '')  
11690:  
11691:         if gap_type == 'missing_baseline':  
11692:             remediation['recommendation'] = "Establecer la base cuantitativa basada en diagnóstico actual"  
11693:         elif gap_type == 'missing_target':  
11694:             remediation['recommendation'] = "Definir meta cuantitativa con horizonte temporal claro"  
11695:         elif gap_type == 'missing_entity':  
11696:             remediation['recommendation'] = "Asignar entidad responsable específica"  
11697:         elif gap_type == 'missing_budget':  
11698:             remediation['recommendation'] = "Asignar presupuesto específico con fuente de financiación"  
11699:         elif gap_type == 'missing_indicator':  
11700:             remediation['recommendation'] = "Definir indicador medible con fórmula de cálculo"  
11701:     else:
```

```
11702:             remediation['recommendation'] = f"Completar {gap_type} segÃ³n estÃ¡ndares DNP"
11703:
11704:             remediations.append(remediation)
11705:
11706:     return remediations
11707:
11708: @calibrated_method("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations")
11709: def generate_recommendations(self, analysis_results: dict[str, Any]) -> list[str]:
11710:     """
11711:         Generate recommendations based on analysis results.
11712:
11713:     Args:
11714:         analysis_results: Results from municipal plan analysis
11715:
11716:     Returns:
11717:         List of actionable recommendations
11718:     """
11719:     recommendations = []
11720:
11721:     # Check financial feasibility
11722:     if analysis_results.get('financial_feasibility', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2379_62", 0.7):
11723:         recommendations.append(
11724:             "Revisar sostenibilidad financiera y diversificar fuentes de financiaciÃ³n"
11725:         )
11726:
11727:     # Check indicator quality
11728:     if analysis_results.get('indicator_quality', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2385_58", 0.7):
11729:         recommendations.append(
11730:             "Mejorar calidad de indicadores: asegurar lÃ¢nea base, meta y fuente de informaciÃ³n"
11731:         )
11732:
11733:     # Check responsibility clarity
11734:     if analysis_results.get('responsibility_clarity', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2391_63", 0.7):
11735:         recommendations.append(
11736:             "Clarificar entidades responsables para cada producto y resultado"
11737:         )
11738:
11739:     # Check temporal consistency
11740:     if analysis_results.get('temporal_consistency', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2397_61", 0.7):
11741:         recommendations.append(
11742:             "Establecer cronograma claro con hitos y plazos definidos"
11743:         )
11744:
11745:     # Check causal coherence
11746:     if analysis_results.get('causal_coherence', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations", "auto_param_L2403_57", 0.7):
11747:         recommendations.append(
11748:             "Fortalecer coherencia causal: vincular productos con resultados e impactos"
11749:         )
11750:
11751:     # PDET-specific recommendations
11752:     if analysis_results.get('is_pdet_municipality', False):
```

```
11753:         if analysis_results.get('pdet_alignment', 0) < ParameterLoaderV2.get("farfan_core.analysis.financiero_vivienda_tablas.PDET_MunicipalPlanAnalyze
r.generate_recommendations", "auto_param_L2410_59", 0.7):
11754:             recommendations.append(
11755:                 "Alinear intervenciones con lineamientos PDET y enfoque territorial"
11756:             )
11757:
11758:     # Generic recommendation if no specific issues
11759:     if not recommendations:
11760:         recommendations.append(
11761:             "El plan cumple con estÃ¡ndares mÃ¡nimos. Considerar monitoreo continuo."
11762:         )
11763:
11764:     return recommendations
11765:
11766: # =====
11767: # UTILIDADES Y HELPERS
11768: # =====
11769:
11770: class PDETAnalysisException(Exception):
11771:     """ExcepciÃ³n personalizada para errores de anÃ;lisis"""
11772:     pass
11773:
11774: def validate_pdf_path(pdf_path: str) -> Path:
11775:     """Valida que el path del PDF exista y sea vÃ¡lido"""
11776:
11777:     path = Path(pdf_path)
11778:
11779:     if not path.exists():
11780:         raise PDETAnalysisException(f"Archivo no encontrado: {pdf_path}")
11781:
11782:     if not path.is_file():
11783:         raise PDETAnalysisException(f"La ruta no es un archivo: {pdf_path}")
11784:
11785:     if path.suffix.lower() != '.pdf':
11786:         raise PDETAnalysisException(f"El archivo debe ser PDF, encontrado: {path.suffix}")
11787:
11788:     return path
11789:
11790: def setup_logging(log_level: str = 'INFO') -> None:
11791:     """Configura logging para el anÃ;lisis"""
11792:
11793:     import logging
11794:
11795:     logging.basicConfig(
11796:         level=getattr(logging, log_level.upper()),
11797:         format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
11798:         handlers=[
11799:             logging.StreamHandler(),
11800:             logging.FileHandler('pdet_analysis.log', encoding='utf-8')
11801:         ]
11802:     )
11803:
11804: # =====
11805: # EJEMPLO DE USO
11806: # =====
11807:
```

```
11808:     async def main_example() -> None:
11809:         """
11810:             Ejemplo de uso del analizador
11811:
11812:             REQUISITOS PREVIOS:
11813:                 1. Instalar dependencias: pip install -r requirements.txt
11814:                 2. Descargar modelo SpaCy: python -m spacy download es_dep_news_trf
11815:                 3. Tener GPU disponible (opcional pero recomendado)
11816:             """
11817:
11818:             # Configurar logging
11819:             setup_logging('INFO')
11820:
11821:             # Inicializar analizador
11822:             analyzer = PDET MunicipalPlanAnalyzer(
11823:                 use_gpu=True,
11824:                 language='es',
11825:                 confidence_threshold = ParameterLoaderV2.get("farfan_core.analysis.financiero_viability_tablas.PDET MunicipalPlanAnalyzer.generate_recommendations",
11826: "confidence_threshold", 0.7) # Refactored
11827:             )
11828:
11829:             # Ruta al PDF del Plan de Desarrollo Municipal
11830:             pdf_path = "path/to/plan_desarrollo_municipal.pdf"
11831:
11832:             try:
11833:                 # Validar archivo
11834:                 validate_pdf_path(pdf_path)
11835:
11836:                 # Ejecutar análisis completo
11837:                 results = await analyzer.analyze_municipal_plan(
11838:                     pdf_path=pdf_path,
11839:                     output_dir="outputs/analisis_pdm"
11840:                 )
11841:
11842:                 # Acceder a resultados específicos
11843:                 print("\nδ\237\223\212 RESULTADOS PRINCIPALES:")
11844:                 print(f" Score de Calidad: {results['quality_score']['overall_score']:.2f}/10")
11845:                 print(f" Presupuesto Total: ${results['financial_analysis']['total_budget']:.0f}")
11846:                 print(f" Efectos Causales Identificados: {len(results['causal_effects'])}")
11847:                 print(f" Escenarios Contrafactuals: {len(results['counterfactuals'])}")
11848:
11849:             except PDETEAnalysisException as e:
11850:                 print(f"\235\214 Error de análisis: {e}")
11851:             except Exception as e:
11852:                 print(f"\235\214 Error inesperado: {e}")
11853:                 raise
11854:
11855:
11856: =====
11857: FILE: src/farfan_pipeline/analysis/graph_metrics_fallback.py
11858: =====
11859:
11860: """
11861: Graph metrics computation with NetworkX fallback handling.
11862:
```

```
11863: This module provides graph metrics computation with graceful degradation
11864: when NetworkX is unavailable. It integrates with the runtime configuration
11865: system to emit proper observability signals.
11866: """
11867:
11868: import logging
11869: from typing import Any, Optional
11870:
11871: from farfan_pipeline.core.runtime_config import RuntimeConfig, get_runtime_config
11872: from farfan_pipeline.core.contracts.runtime_contracts import (
11873:     GraphMetricsInfo,
11874:     FallbackCategory,
11875: )
11876: from farfan_pipeline.core.observability.structured_logging import log_fallback
11877: from farfan_pipeline.core.observability.metrics import increment_graph_metrics_skipped
11878:
11879: logger = logging.getLogger(__name__)
11880:
11881:
11882: def check_networkx_available() -> bool:
11883:     """
11884:         Check if NetworkX is available for graph metrics computation.
11885:
11886:     Returns:
11887:         True if NetworkX is available, False otherwise
11888:     """
11889:     try:
11890:         import networkx
11891:         return True
11892:     except ImportError:
11893:         return False
11894:
11895:
11896: def compute_graph_metrics_with_fallback(
11897:     graph_data: Any,
11898:     runtime_config: Optional[RuntimeConfig] = None,
11899:     document_id: Optional[str] = None,
11900: ) -> tuple[dict[str, Any], GraphMetricsInfo]:
11901:     """
11902:         Compute graph metrics with NetworkX fallback handling.
11903:
11904:     Args:
11905:         graph_data: Graph data structure (e.g., edge list, adjacency matrix)
11906:         runtime_config: Optional runtime configuration (uses global if None)
11907:         document_id: Optional document identifier for logging
11908:
11909:     Returns:
11910:         Tuple of (metrics_dict, GraphMetricsInfo manifest)
11911:
11912:     Example:
11913:         >>> metrics, info = compute_graph_metrics_with_fallback(edge_list)
11914:         >>> if info.computed:
11915:             ...     print(f"Centrality: {metrics['centrality']}")
11916:             ... else:
11917:                 ...     print(f"Skipped: {info.reason}")
11918:     """
```

```
11919:     if runtime_config is None:
11920:         runtime_config = get_runtime_config()
11921:
11922:     networkx_available = check_networkx_available()
11923:
11924:     if networkx_available:
11925:         try:
11926:             import networkx as nx
11927:
11928:                 # Convert graph_data to NetworkX graph
11929:                 # This is a placeholder - actual implementation depends on graph_data format
11930:                 if isinstance(graph_data, list):
11931:                     # Assume edge list format: [(source, target), ...]
11932:                     G = nx.Graph()
11933:                     G.add_edges_from(graph_data)
11934:                 elif isinstance(graph_data, dict):
11935:                     # Assume adjacency dict format
11936:                     G = nx.from_dict_of_lists(graph_data)
11937:                 else:
11938:                     raise ValueError(f"Unsupported graph_data type: {type(graph_data)}")
11939:
11940:                 # Compute graph metrics
11941:                 metrics = {
11942:                     'num_nodes': G.number_of_nodes(),
11943:                     'num_edges': G.number_of_edges(),
11944:                     'density': nx.density(G),
11945:                     'avg_clustering': nx.average_clustering(G) if G.number_of_nodes() > 0 else 0.0,
11946:                     'num_components': nx.number_connected_components(G),
11947:                 }
11948:
11949:                 # Compute centrality if graph is not too large
11950:                 if G.number_of_nodes() < 1000:
11951:                     metrics['degree_centrality'] = nx.degree_centrality(G)
11952:                     metrics['betweenness_centrality'] = nx.betweenness_centrality(G)
11953:
11954:                     logger.info(f"Graph metrics computed: {metrics['num_nodes']} nodes, {metrics['num_edges']} edges")
11955:
11956:                     graph_info = GraphMetricsInfo(
11957:                         computed=True,
11958:                         networkx_available=True,
11959:                         reason=None
11960:                     )
11961:
11962:                     return metrics, graph_info
11963:
11964:                 except Exception as e:
11965:                     # NetworkX available but computation failed
11966:                     logger.error(f"Graph metrics computation failed: {e}")
11967:
11968:                     reason = f"NetworkX computation error: {str(e)}"
11969:                     graph_info = GraphMetricsInfo(
11970:                         computed=False,
11971:                         networkx_available=True,
11972:                         reason=reason
11973:                     )
11974:
```

```
11975:         # Emit structured log and metrics (Category B: Quality degradation)
11976:         log_fallback(
11977:             component='graph_metrics',
11978:             subsystem='analysis',
11979:             fallback_category=FallbackCategory.B,
11980:             fallback_mode='computation_error',
11981:             reason=reason,
11982:             runtime_mode=runtime_config.mode,
11983:             document_id=document_id,
11984:         )
11985:
11986:         increment_graph_metrics_skipped(
11987:             reason='computation_error',
11988:             runtime_mode=runtime_config.mode,
11989:         )
11990:
11991:         # Return empty metrics
11992:         return {}, graph_info
11993:
11994:     else:
11995:         # NetworkX not available - graceful degradation
11996:         reason = "NetworkX not available - graph metrics skipped"
11997:         logger.warning(reason)
11998:
11999:         graph_info = GraphMetricsInfo(
12000:             computed=False,
12001:             networkx_available=False,
12002:             reason=reason
12003:         )
12004:
12005:         # Emit structured log and metrics (Category B: Quality degradation)
12006:         log_fallback(
12007:             component='graph_metrics',
12008:             subsystem='analysis',
12009:             fallback_category=FallbackCategory.B,
12010:             fallback_mode='networkx_unavailable',
12011:             reason=reason,
12012:             runtime_mode=runtime_config.mode,
12013:             document_id=document_id,
12014:         )
12015:
12016:         increment_graph_metrics_skipped(
12017:             reason='networkx_unavailable',
12018:             runtime_mode=runtime_config.mode,
12019:         )
12020:
12021:         # Return empty metrics
12022:         return {}, graph_info
12023:
12024:
12025: def compute_basic_graph_stats(graph_data: Any) -> dict[str, Any]:
12026:     """
12027:     Compute basic graph statistics without NetworkX.
12028:
12029:     This is a lightweight fallback that computes basic stats
12030:     without requiring NetworkX.
```

```
12031:  
12032:     Args:  
12033:         graph_data: Graph data (edge list or adjacency dict)  
12034:  
12035:     Returns:  
12036:         Dictionary with basic graph statistics  
12037:     """  
12038:     if isinstance(graph_data, list):  
12039:         # Edge list format  
12040:         nodes = set()  
12041:         for edge in graph_data:  
12042:             if len(edge) >= 2:  
12043:                 nodes.add(edge[0])  
12044:                 nodes.add(edge[1])  
12045:  
12046:         return {  
12047:             'num_nodes': len(nodes),  
12048:             'num_edges': len(graph_data),  
12049:             'method': 'basic_stats_no_networkx'  
12050:         }  
12051:  
12052:     elif isinstance(graph_data, dict):  
12053:         # Adjacency dict format  
12054:         num_edges = sum(len(neighbors) for neighbors in graph_data.values())  
12055:  
12056:         return {  
12057:             'num_nodes': len(graph_data),  
12058:             'num_edges': num_edges // 2, # Undirected graph  
12059:             'method': 'basic_stats_no_networkx'  
12060:         }  
12061:  
12062:     else:  
12063:         return {  
12064:             'num_nodes': 0,  
12065:             'num_edges': 0,  
12066:             'method': 'unknown_format'  
12067:         }  
12068:  
12069:  
12070:  
12071: ======  
12072: FILE: src/farfan_pipeline/analysis/macro_prompts.py  
12073: ======  
12074:  
12075: # macro_prompts.py  
12076: """  
12077: Macro Prompts for MACRO-Level Analysis  
12078: ======  
12079:  
12080: This module implements 5 strategic macro-level analysis prompts:  
12081: 1. Coverage & Structural Gap Stressor - Evaluates dimensional/cluster coverage  
12082: 2. Inter-Level Contradiction Scan - Detects microâ\206\224mesoâ\206\224macro contradictions  
12083: 3. Bayesian Portfolio Composer - Integrates posteriors into global portfolio  
12084: 4. Roadmap Optimizer - Generates sequenced 0-3m / 3-6m / 6-12m roadmap  
12085: 5. Peer Normalization & Confidence Scaling - Adjusts classification vs peers  
12086:
```

```
12087: Author: Integration Team
12088: Version: 1.0.0
12089: Python: 3.10+
12090: """
12091:
12092: import logging
12093: import statistics
12094: import warnings
12095: from dataclasses import asdict, dataclass, field
12096: from typing import Any
12097:
12098: from farfan_pipeline.core.parameters import ParameterLoaderV2
12099:
12100: # Import runtime error fixes for defensive programming
12101: from farfan_pipeline.utils.runtime_error_fixes import ensure_list_return
12102:
12103: logger = logging.getLogger(__name__)
12104:
12105: # =====
12106: # DATA STRUCTURES FOR MACRO PROMPTS
12107: # =====
12108:
12109: @dataclass
12110: class CoverageAnalysis:
12111:     """Output from Coverage & Structural Gap Stressor"""
12112:     coverage_index: float # Weighted average coverage (0.0-1.0)
12113:     degraded_confidence: float | None # Adjusted confidence if coverage low
12114:     predictive_uplift: dict[str, float] # Expected improvement if gaps filled
12115:     dimension_coverage: dict[str, float] # D1-D6 coverage percentages
12116:     policy_area_coverage: dict[str, float] # P1-P10 coverage percentages
12117:     critical_dimensions_below_threshold: list[str] # Dimensions needing attention
12118:     metadata: dict[str, Any] = field(default_factory=dict)
12119:
12120: @dataclass
12121: class ContradictionReport:
12122:     """Output from Inter-Level Contradiction Scan"""
12123:     contradictions: list[dict[str, Any]] # List of detected contradictions
12124:     suggested_actions: list[dict[str, str]] # Actions to resolve contradictions
12125:     consistency_score: float # 0.0-1.0 overall consistency
12126:     micro_meso_alignment: float # 0.0-1.0 micro\206\224meso alignment
12127:     meso_macro_alignment: float # 0.0-1.0 meso\206\224macro alignment
12128:     metadata: dict[str, Any] = field(default_factory=dict)
12129:
12130: @dataclass
12131: class BayesianPortfolio:
12132:     """Output from Bayesian Portfolio Composer"""
12133:     prior_global: float # Global prior (weighted meso average)
12134:     penalties_applied: dict[str, float] # Coverage, dispersion, contradiction penalties
12135:     posterior_global: float # Adjusted global posterior
12136:     var_global: float # Global variance
12137:     confidence_interval: tuple[float, float] # 95% CI
12138:     metadata: dict[str, Any] = field(default_factory=dict)
12139:
12140: @dataclass
12141: class ImplementationRoadmap:
12142:     """Output from Roadmap Optimizer
```

```
12143:  
12144:     DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.  
12145:         Metadata:  
12146:             - creation_date: 2024-01-15  
12147:             - last_used_date: null  
12148:             - intended_phase: future_phase_7_9  
12149:             - deprecation_status: candidate_for_removal_2025Q2  
12150:         """  
12151:         phases: list[dict[str, Any]] # 0-3m, 3-6m, 6-12m phases  
12152:         total_expected_uplift: float # Total expected improvement  
12153:         critical_path: list[str] # Critical dependency chain  
12154:         resource_requirements: dict[str, Any] # Estimated resources per phase  
12155:         metadata: dict[str, Any] = field(default_factory=dict)  
12156:  
12157:     def __post_init__(self) -> None:  
12158:         warnings.warn(  
12159:             "ImplementationRoadmap is deprecated and scheduled for removal in 2025Q2. "  
12160:             "No usage detected as of February 2025. Intended for future_phase_7_9. "  
12161:             "Metadata: creation_date=2024-01-15, last_used_date=null, "  
12162:             "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",  
12163:             DeprecationWarning,  
12164:             stacklevel=2  
12165:         )  
12166:  
12167: @dataclass  
12168: class PeerNormalization:  
12169:     """Output from Peer Normalization & Confidence Scaling  
12170:  
12171:     DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.  
12172:         Metadata:  
12173:             - creation_date: 2024-01-15  
12174:             - last_used_date: null  
12175:             - intended_phase: future_phase_7_9  
12176:             - deprecation_status: candidate_for_removal_2025Q2  
12177:         """  
12178:         z_scores: dict[str, float] # Z-scores by policy area  
12179:         adjusted_confidence: float # Adjusted confidence based on peer comparison  
12180:         peer_position: str # "above_average", "average", "below_average"  
12181:         outlier_areas: list[str] # Policy areas >2 SD from mean  
12182:         metadata: dict[str, Any] = field(default_factory=dict)  
12183:  
12184:     def __post_init__(self) -> None:  
12185:         warnings.warn(  
12186:             "PeerNormalization is deprecated and scheduled for removal in 2025Q2. "  
12187:             "No usage detected as of February 2025. Intended for future_phase_7_9. "  
12188:             "Metadata: creation_date=2024-01-15, last_used_date=null, "  
12189:             "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",  
12190:             DeprecationWarning,  
12191:             stacklevel=2  
12192:         )  
12193:  
12194: # ======  
12195: # MACRO PROMPT 1: COVERAGE & STRUCTURAL GAP STRESSOR  
12196: # ======  
12197:  
12198: class CoverageGapStressor:
```

```
12199: """
12200:     ROLE: Structural Integrity Auditor [systems design]
12201:     GOAL: Evaluar si la ausencia de clusters o dimensiones erosiona la validez del score macro.
12202:
12203:     INPUTS:
12204:         - convergence_by_dimension
12205:         - missing_clusters
12206:         - dimension_coverage: {D1..D6: % preguntas respondidas}
12207:         - policy_area_coverage: {P#: %}
12208:
12209:     MANDATES:
12210:         - Calcular coverage_index (media ponderada)
12211:         - Si dimension_coverage < 0.204 en alguna dimensiÃ³n crÃ¡tica (D3, D6) degradar global_confidence
12212:         - Simular impacto si se completara el cluster faltante (predictive uplift)
12213:
12214:     OUTPUT:
12215:     JSON {coverage_index, degraded_confidence, predictive_uplift}
12216:
12217: DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
12218: Metadata:
12219:     - creation_date: 2024-01-15
12220:     - last_used_date: null
12221:     - intended_phase: future_phase_7_9
12222:     - deprecation_status: candidate_for_removal_2025Q2
12223: """
12224:
12225: def __init__(
12226:     self,
12227:     critical_dimensions: list[str] | None = None,
12228:     dimension_weights: dict[str, float] | None = None,
12229:     coverage_threshold: float = 0.70
12230: ) -> None:
12231:     """
12232:         Initialize Coverage Gap Stressor
12233:
12234:     Args:
12235:         critical_dimensions: List of critical dimensions (default: D3, D6)
12236:         dimension_weights: Weights for each dimension (default: equal)
12237:         coverage_threshold: Minimum acceptable coverage (default: 0.70)
12238:     """
12239:     warnings.warn(
12240:         "CoverageGapStressor is deprecated and scheduled for removal in 2025Q2. "
12241:         "No usage detected as of February 2025. Intended for future_phase_7_9. "
12242:         "Metadata: creation_date=2024-01-15, last_used_date=null, "
12243:         "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
12244:         DeprecationWarning,
12245:         stacklevel=2
12246:     )
12247:     self.critical_dimensions = critical_dimensions or ["D3", "D6"]
12248:     self.dimension_weights = dimension_weights or {
12249:         f"D{i}": 1.0/6.0 for i in range(1, 7)
12250:     }
12251:     self.coverage_threshold = coverage_threshold
12252:     logger.info(f"Cov  
erageGapStressor initialized with threshold={coverage_threshold}")
12253:
12254:     def evaluate(
```

```
12255:         self,
12256:         convergence_by_dimension: dict[str, float],
12257:         missing_clusters: list[str],
12258:         dimension_coverage: dict[str, float],
12259:         policy_area_coverage: dict[str, float],
12260:         baseline_confidence: float = 1.0
12261:     ) -> CoverageAnalysis:
12262:     """
12263:     Evaluate coverage and structural gaps
12264:
12265:     Args:
12266:         convergence_by_dimension: Convergence scores by dimension
12267:         missing_clusters: List of missing cluster names
12268:         dimension_coverage: Coverage percentage by dimension
12269:         policy_area_coverage: Coverage percentage by policy area
12270:         baseline_confidence: Starting confidence level (0.0-1.0)
12271:
12272:     Returns:
12273:         CoverageAnalysis with complete gap assessment
12274:     """
12275:     # Calculate weighted coverage index
12276:     coverage_index = self._calculate_coverage_index(dimension_coverage)
12277:
12278:     # Check critical dimensions
12279:     critical_below_threshold = self._identify_critical_gaps(dimension_coverage)
12280:
12281:     # Degrade confidence if critical gaps exist
12282:     degraded_confidence = self._degrade_confidence(
12283:         baseline_confidence,
12284:         critical_below_threshold,
12285:         coverage_index
12286:     )
12287:
12288:     # Simulate predictive uplift
12289:     predictive_uplift = self._simulate_uplift(
12290:         missing_clusters,
12291:         dimension_coverage,
12292:         convergence_by_dimension
12293:     )
12294:
12295:     return CoverageAnalysis(
12296:         coverage_index=coverage_index,
12297:         degraded_confidence=degraded_confidence,
12298:         predictive_uplift=predictive_uplift,
12299:         dimension_coverage=dimension_coverage,
12300:         policy_area_coverage=policy_area_coverage,
12301:         critical_dimensions_below_threshold=critical_below_threshold,
12302:         metadata={
12303:             "missing_clusters": missing_clusters,
12304:             "threshold_used": self.coverage_threshold,
12305:             "critical_dimensions": self.critical_dimensions
12306:         }
12307:     )
12308:
12309:     def _calculate_coverage_index(
12310:         self,
```

```

12311:     dimension_coverage: dict[str, float]
12312: ) -> float:
12313:     """Calculate weighted average coverage index"""
12314:     total_weight = 0.0
12315:     weighted_sum = 0.0
12316:
12317:     for dim, coverage in dimension_coverage.items():
12318:         weight = self.dimension_weights.get(dim, 0.0)
12319:         weighted_sum += coverage * weight
12320:         total_weight += weight
12321:
12322:     if total_weight == 0:
12323:         return 0.0
12324:
12325:     return weighted_sum / total_weight
12326:
12327: def _identify_critical_gaps(
12328:     self,
12329:     dimension_coverage: dict[str, float]
12330: ) -> list[str]:
12331:     """Identify critical dimensions below threshold"""
12332:     critical_gaps = []
12333:
12334:     for dim in self.critical_dimensions:
12335:         if dim in dimension_coverage:
12336:             if dimension_coverage[dim] < self.coverage_threshold:
12337:                 critical_gaps.append(dim)
12338:
12339:     return critical_gaps
12340:
12341: def _degrade_confidence(
12342:     self,
12343:     baseline_confidence: float,
12344:     critical_gaps: list[str],
12345:     coverage_index: float
12346: ) -> float:
12347:     """Degrade confidence based on structural gaps"""
12348:     degraded = baseline_confidence
12349:
12350:     # Penalty for each critical gap
12351:     for _ in critical_gaps:
12352:         degraded *= 0.85 # 15% penalty per critical gap
12353:
12354:     # Additional penalty if overall coverage is low
12355:     if coverage_index < self.coverage_threshold:
12356:         gap_severity = (self.coverage_threshold - coverage_index) / self.coverage_threshold
12357:         degraded *= (1.0 - gap_severity * 0.3) # Up to 30% additional penalty
12358:
12359:     return max(0.0, min(1.0, degraded))
12360:
12361: def _simulate_uplift(
12362:     self,
12363:     missing_clusters: list[str],
12364:     dimension_coverage: dict[str, float],
12365:     convergence_by_dimension: dict[str, float]
12366: ) -> dict[str, float]:

```

```
12367:     """Simulate impact if missing clusters were completed"""
12368:     uplift = {}
12369:
12370:     # Estimate uplift for each missing cluster
12371:     for cluster in missing_clusters:
12372:         # Assume cluster completion would improve coverage by 10-20%
12373:         estimated_improvement = 0.15
12374:         uplift[cluster] = estimated_improvement
12375:
12376:     # Estimate dimension-level uplift
12377:     for dim, coverage in dimension_coverage.items():
12378:         if coverage < 1.0:
12379:             gap = 1.0 - coverage
12380:             convergence = convergence_by_dimension.get(dim, 0.5)
12381:             # Higher convergence suggests more potential uplift
12382:             potential_uplift = gap * convergence * 0.7
12383:             uplift[f"{dim}_completion"] = potential_uplift
12384:
12385:     return uplift
12386:
12387: # =====
12388: # MACRO PROMPT 2: INTER-LEVEL CONTRADICTION SCAN
12389: # =====
12390:
12391: class ContradictionScanner:
12392:     """
12393:         ROLE: Consistency Inspector [data governance]
12394:         GOAL: Detectar contradicciones micro\206\224meso\206\224macro.
12395:
12396:         INPUTS:
12397:         - micro_claims (extraÃ±o de MicroLevelAnswer.evidence)
12398:         - meso_summary_signals
12399:         - macro_narratives (borrador)
12400:
12401:         MANDATES:
12402:         - Alinear claims por entidad/tema/dimensiÃ³n
12403:         - Marcar contradicciÃ³n si macro afirma X y â\211k micro niegan X con posterior â\211y Î,
12404:         - Sugerir correcciÃ³n: "rephrase / downgrade confidence / request re-execution"
12405:
12406:         OUTPUT:
12407:         JSON {contradictions[], suggested_actions}
12408:
12409:         DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
12410:         Metadata:
12411:             - creation_date: 2024-01-15
12412:             - last_used_date: null
12413:             - intended_phase: future_phase_7_9
12414:             - deprecation_status: candidate_for_removal_2025Q2
12415: """
12416:
12417:     def __init__(
12418:         self,
12419:         contradiction_threshold: int = 3,
12420:         posterior_threshold: float = 0.7
12421:     ) -> None:
12422:         """
```

```
12423:     Initialize Contradiction Scanner
12424:
12425:     Args:
12426:         contradiction_threshold: Min number of micro claims to flag contradiction
12427:         posterior_threshold: Min posterior confidence to consider claim valid
12428:     """
12429:     warnings.warn(
12430:         "ContradictionScanner is deprecated and scheduled for removal in 2025Q2. "
12431:         "No usage detected as of February 2025. Intended for future_phase_7_9. "
12432:         "Metadata: creation_date=2024-01-15, last_used_date=null, "
12433:         "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
12434:         DeprecationWarning,
12435:         stacklevel=2
12436:     )
12437:     self.k = contradiction_threshold
12438:     self.theta = posterior_threshold
12439:     logger.info(f"ContradictionScanner initialized (k={self.k}, θ={self.theta})")
12440:
12441:     def scan(
12442:         self,
12443:         micro_claims: list[dict[str, Any]],
12444:         meso_summary_signals: dict[str, Any],
12445:         macro_narratives: dict[str, Any]
12446:     ) -> ContradictionReport:
12447:     """
12448:         Scan for contradictions across levels
12449:
12450:         Args:
12451:             micro_claims: List of micro-level claims with evidence
12452:             meso_summary_signals: Meso-level summary signals
12453:             macro_narratives: Macro-level narrative statements
12454:
12455:         Returns:
12456:             ContradictionReport with detected issues and suggested actions
12457:     """
12458:     # Align claims by entity/theme/dimension
12459:     aligned_claims = self._align_claims(micro_claims, meso_summary_signals, macro_narratives)
12460:
12461:     # Detect contradictions (defensive: ensure returns list)
12462:     contradictions = ensure_list_return(self._detect_contradictions(aligned_claims))
12463:
12464:     # Generate suggested actions
12465:     suggested_actions = self._generate_actions(contradictions)
12466:
12467:     # Calculate consistency scores
12468:     consistency_score = self._calculate_consistency(contradictions, len(micro_claims))
12469:     micro_meso_alignment = self._calculate_alignment(micro_claims, meso_summary_signals)
12470:     meso_macro_alignment = self._calculate_alignment(
12471:         [meso_summary_signals],
12472:         macro_narratives
12473:     )
12474:
12475:     return ContradictionReport(
12476:         contradictions=contradictions,
12477:         suggested_actions=suggested_actions,
12478:         consistency_score=consistency_score,
```

```

12479:         micro_meso_alignment=micro_meso_alignment,
12480:         meso_macro_alignment=meso_macro_alignment,
12481:         metadata={
12482:             "total_micro_claims": len(micro_claims),
12483:             "contradiction_threshold": self.k,
12484:             "posterior_threshold": self.theta
12485:         }
12486:     )
12487:
12488:     def _align_claims(
12489:         self,
12490:         micro_claims: list[dict[str, Any]],
12491:         meso_summary_signals: dict[str, Any],
12492:         macro_narratives: dict[str, Any]
12493:     ) -> dict[str, dict[str, list[Any]]]:
12494:         """Align claims by entity/theme/dimension"""
12495:         aligned = {
12496:             "micro": {},
12497:             "meso": {},
12498:             "macro": {}
12499:         }
12500:
12501:         # Group micro claims by dimension
12502:         for claim in micro_claims:
12503:             dimension = claim.get("dimension", "unknown")
12504:             if dimension not in aligned["micro"]:
12505:                 aligned["micro"][dimension] = []
12506:                 aligned["micro"][dimension].append(claim)
12507:
12508:         # Group meso signals by dimension
12509:         for key, value in meso_summary_signals.items():
12510:             if key.startswith("D") and len(key) == 2:
12511:                 aligned["meso"][key] = value
12512:
12513:         # Group macro narratives
12514:         aligned["macro"] = macro_narratives
12515:
12516:         return aligned
12517:
12518:     def _detect_contradictions(
12519:         self,
12520:         aligned_claims: dict[str, dict[str, Any]]
12521:     ) -> list[dict[str, Any]]:
12522:         """Detect contradictions across levels"""
12523:         contradictions = []
12524:
12525:         # Check each dimension/theme
12526:         for dimension in aligned_claims.get("micro", {}):
12527:             micro_claims = aligned_claims["micro"].get(dimension, [])
12528:             aligned_claims["meso"].get(dimension, {})
12529:             macro_narrative = aligned_claims["macro"].get(dimension, {})
12530:
12531:             # Count claims that contradict macro narrative
12532:             contradicting_claims = []
12533:             for claim in micro_claims:
12534:                 if self._is_contradictory(claim, macro_narrative):

```

```

12535:             posterior = claim.get("posterior", 0.0)
12536:             if posterior >= self.theta:
12537:                 contradicting_claims.append(claim)
12538:
12539:             # Flag if threshold exceeded
12540:             if len(contradicting_claims) >= self.k:
12541:                 contradictions.append({
12542:                     "dimension": dimension,
12543:                     "type": "micro_macro_contradiction",
12544:                     "contradicting_claims": len(contradicting_claims),
12545:                     "threshold": self.k,
12546:                     "details": contradicting_claims[:5] # Sample
12547:                 })
12548:
12549:             return contradictions
12550:
12551:     def _is_contradictory(
12552:         self,
12553:         claim: dict[str, Any],
12554:         narrative: dict[str, Any]
12555:     ) -> bool:
12556:         """Check if claim contradicts narrative"""
12557:         # Simple heuristic: if claim score is low but narrative is positive
12558:         claim_score = claim.get("score", 0.0)
12559:         narrative_score = narrative.get("score", 0.5)
12560:
12561:         # Contradiction if scores differ significantly
12562:         return abs(claim_score - narrative_score) > 0.4
12563:
12564:     def _generate_actions(
12565:         self,
12566:         contradictions: list[dict[str, Any]]
12567:     ) -> list[dict[str, str]]:
12568:         """Generate suggested actions to resolve contradictions"""
12569:         actions = []
12570:
12571:         for contradiction in contradictions:
12572:             dimension = contradiction.get("dimension", "unknown")
12573:             count = contradiction.get("contradicting_claims", 0)
12574:
12575:             if count >= self.k * 2:
12576:                 actions.append({
12577:                     "dimension": dimension,
12578:                     "action": "request_re_execution",
12579:                     "reason": f"{count} micro claims contradict macro narrative"
12580:                 })
12581:             elif count >= self.k:
12582:                 actions.append({
12583:                     "dimension": dimension,
12584:                     "action": "downgrade_confidence",
12585:                     "reason": f"{count} micro claims suggest lower confidence"
12586:                 })
12587:             else:
12588:                 actions.append({
12589:                     "dimension": dimension,
12590:                     "action": "rephrase_narrative",

```

```
12591:             "reason": "Minor inconsistencies detected"
12592:         })
12593:
12594:     return actions
12595:
12596:     def _calculate_consistency(
12597:         self,
12598:         contradictions: list[dict[str, Any]],
12599:         total_claims: int
12600:     ) -> float:
12601:         """Calculate overall consistency score"""
12602:         if total_claims == 0:
12603:             return 1.0
12604:
12605:         total_contradictions = sum(
12606:             c.get("contradicting_claims", 0) for c in contradictions
12607:         )
12608:
12609:         consistency = 1.0 - (total_contradictions / max(total_claims, 1))
12610:         return max(0.0, min(1.0, consistency))
12611:
12612:     def _calculate_alignment(
12613:         self,
12614:         level1_data: Any,
12615:         level2_data: Any
12616:     ) -> float:
12617:         """Calculate alignment between two levels"""
12618:         # Simplified alignment calculation
12619:         # In production, would use semantic similarity, score correlation, etc.
12620:         return 0.85 # Placeholder
12621:
12622: # =====
12623: # MACRO PROMPT 3: BAYESIAN PORTFOLIO COMPOSER
12624: # =====
12625:
12626: class BayesianPortfolioComposer:
12627:     """
12628:         ROLE: Global Bayesian Integrator [causal inference]
12629:         GOAL: Integrar todas las posteriors (micro y meso) en una cartera causal global.
12630:
12631:         INPUTS:
12632:             - meso_posteriors
12633:             - weighting_trace (cluster_weights)
12634:             - macro_reconciliation_penalties
12635:
12636:         MANDATES:
12637:             - Calcular prior_global (media ponderada meso)
12638:             - Aplicar penalties jerárquicos (coverage, dispersion estructural, contradictions)
12639:             - Recalcular posterior_global y varianza
12640:
12641:         OUTPUT:
12642:             JSON {prior_global, penalties_applied, posterior_global, var_global}
12643:
12644:         DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
12645:         Metadata:
12646:             - creation_date: 2024-01-15
```

```
12647:         - last_used_date: null
12648:         - intended_phase: future_phase_7_9
12649:         - deprecation_status: candidate_for_removal_2025Q2
12650:     """
12651:
12652:     def __init__(
12653:         self,
12654:         default_variance: float = 0.05
12655:     ) -> None:
12656:         """
12657:             Initialize Bayesian Portfolio Composer
12658:
12659:             Args:
12660:                 default_variance: Default variance for uncertain estimates
12661:             """
12662:             warnings.warn(
12663:                 "BayesianPortfolioComposer is deprecated and scheduled for removal in 2025Q2. "
12664:                 "No usage detected as of February 2025. Intended for future_phase_7_9. "
12665:                 "Metadata: creation_date=2024-01-15, last_used_date=null, "
12666:                 "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
12667:                 DeprecationWarning,
12668:                 stacklevel=2
12669:             )
12670:             self.default_variance = default_variance
12671:             logger.info("BayesianPortfolioComposer initialized")
12672:
12673:     def compose(
12674:         self,
12675:         meso_posteriors: dict[str, float],
12676:         cluster_weights: dict[str, float],
12677:         reconciliation_penalties: dict[str, float] | None = None
12678:     ) -> BayesianPortfolio:
12679:         """
12680:             Compose global Bayesian portfolio from meso posteriors
12681:
12682:             Args:
12683:                 meso_posteriors: Posterior probabilities by cluster/dimension
12684:                 cluster_weights: Weights for each cluster
12685:                 reconciliation_penalties: Optional penalties (coverage, dispersion, contradictions)
12686:
12687:             Returns:
12688:                 BayesianPortfolio with integrated global estimate
12689:             """
12690:             # Calculate weighted prior
12691:             prior_global = self._calculate_weighted_prior(meso_posteriors, cluster_weights)
12692:
12693:             # Apply hierarchical penalties
12694:             penalties = reconciliation_penalties or {}
12695:             penalties_applied = self._apply_penalties(prior_global, penalties)
12696:
12697:             # Calculate posterior and variance
12698:             posterior_global = self._calculate_posterior(prior_global, penalties_applied)
12699:             var_global = self._calculate_variance(meso_posteriors, cluster_weights, penalties_applied)
12700:
12701:             # Calculate 95% confidence interval
12702:             ci = self._calculate_confidence_interval(posterior_global, var_global)
```

```
12703:  
12704:         return BayesianPortfolio(  
12705:             prior_global=prior_global,  
12706:             penalties_applied=penalties_applied,  
12707:             posterior_global=posterior_global,  
12708:             var_global=var_global,  
12709:             confidence_interval=ci,  
12710:             metadata={  
12711:                 "num_clusters": len(meso_posteriors),  
12712:                 "total_weight": sum(cluster_weights.values())  
12713:             }  
12714:         )  
12715:  
12716:     def _calculate_weighted_prior(  
12717:         self,  
12718:         meso_posteriors: dict[str, float],  
12719:         cluster_weights: dict[str, float]  
12720:     ) -> float:  
12721:         """Calculate weighted prior from meso posteriors"""  
12722:         total_weight = sum(cluster_weights.values())  
12723:         if total_weight == 0:  
12724:             return 0.5 # Neutral prior  
12725:  
12726:         weighted_sum = 0.0  
12727:         for cluster, posterior in meso_posteriors.items():  
12728:             weight = cluster_weights.get(cluster, 0.0)  
12729:             weighted_sum += posterior * weight  
12730:  
12731:         return weighted_sum / total_weight  
12732:  
12733:     def _apply_penalties(  
12734:         self,  
12735:         prior: float,  
12736:         penalties: dict[str, float]  
12737:     ) -> dict[str, float]:  
12738:         """Apply hierarchical penalties"""  
12739:         applied = {}  
12740:  
12741:         # Coverage penalty  
12742:         coverage_penalty = penalties.get("coverage", 0.0)  
12743:         applied["coverage"] = coverage_penalty  
12744:  
12745:         # Structural dispersion penalty  
12746:         dispersion_penalty = penalties.get("dispersion", 0.0)  
12747:         applied["dispersion"] = dispersion_penalty  
12748:  
12749:         # Contradiction penalty  
12750:         contradiction_penalty = penalties.get("contradictions", 0.0)  
12751:         applied["contradictions"] = contradiction_penalty  
12752:  
12753:         return applied  
12754:  
12755:     def _calculate_posterior(  
12756:         self,  
12757:         prior: float,  
12758:         penalties: dict[str, float]
```

```
12759:     ) -> float:
12760:         """Calculate posterior after applying penalties"""
12761:         posterior = prior
12762:
12763:         # Apply each penalty multiplicatively
12764:         for _penalty_name, penalty_value in penalties.items():
12765:             posterior *= (1.0 - penalty_value)
12766:
12767:         return max(0.0, min(1.0, posterior))
12768:
12769:     def _calculate_variance(
12770:         self,
12771:         meso_posteriors: dict[str, float],
12772:         cluster_weights: dict[str, float],
12773:         penalties: dict[str, float]
12774:     ) -> float:
12775:         """Calculate global variance"""
12776:         if len(meso_posteriors) < 2:
12777:             return self.default_variance
12778:
12779:         # Calculate weighted variance
12780:         mean = self._calculate_weighted_prior(meso_posteriors, cluster_weights)
12781:         total_weight = sum(cluster_weights.values())
12782:
12783:         if total_weight == 0:
12784:             return self.default_variance
12785:
12786:         weighted_sq_diff = 0.0
12787:         for cluster, posterior in meso_posteriors.items():
12788:             weight = cluster_weights.get(cluster, 0.0)
12789:             sq_diff = (posterior - mean) ** 2
12790:             weighted_sq_diff += weight * sq_diff
12791:
12792:         variance = weighted_sq_diff / total_weight
12793:
12794:         # Increase variance based on penalties
12795:         penalty_factor = 1.0 + sum(penalties.values())
12796:         adjusted_variance = variance * penalty_factor
12797:
12798:         return adjusted_variance
12799:
12800:     def _calculate_confidence_interval(
12801:         self,
12802:         posterior: float,
12803:         variance: float,
12804:         confidence: float = 0.95
12805:     ) -> tuple[float, float]:
12806:         """Calculate confidence interval (assumes normal distribution)"""
12807:         # For 95% CI, z-score ≈ 1.96
12808:         z_score = 1.96
12809:         margin = z_score * (variance ** 0.5)
12810:
12811:         lower = max(0.0, posterior - margin)
12812:         upper = min(1.0, posterior + margin)
12813:
12814:         return (lower, upper)
```



```

12871:         dependency_graph: Gap ID -> list of prerequisite gap IDs
12872:         effort_estimates: Gap ID -> effort estimate (person-months)
12873:         impact_scores: Gap ID -> expected impact (ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L702_54", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L702_58", 1.0))
12874:
12875:     Returns:
12876:         ImplementationRoadmap with phased action plan
12877:     """
12878:     # Calculate impact/effort ratios
12879:     prioritized_gaps = self._prioritize_gaps(
12880:         critical_gaps,
12881:         effort_estimates,
12882:         impact_scores
12883:     )
12884:
12885:     # Assign to time windows respecting dependencies
12886:     phases = self._assign_phases(
12887:         prioritized_gaps,
12888:         dependency_graph,
12889:         effort_estimates
12890:     )
12891:
12892:     # Calculate expected uplift per phase
12893:     total_uplift = self._calculate_total_uplift(phases, impact_scores)
12894:
12895:     # Identify critical path
12896:     critical_path = self._identify_critical_path(dependency_graph, impact_scores)
12897:
12898:     # Estimate resource requirements
12899:     resources = self._estimate_resources(phases, effort_estimates)
12900:
12901:     return ImplementationRoadmap(
12902:         phases=phases,
12903:         total_expected_uplift=total_uplift,
12904:         critical_path=critical_path,
12905:         resource_requirements=resources,
12906:         metadata={
12907:             "total_gaps": len(critical_gaps),
12908:             "total_effort": sum(effort_estimates.values())
12909:         }
12910:     )
12911:
12912:     def _prioritize_gaps(
12913:         self,
12914:         gaps: list[dict[str, Any]],
12915:         effort_estimates: dict[str, float],
12916:         impact_scores: dict[str, float]
12917:     ) -> list[dict[str, Any]]:
12918:         """Prioritize gaps by impact/effort ratio"""
12919:         prioritized = []
12920:
12921:         for gap in gaps:
12922:             gap_id = gap.get("id", "unknown")
12923:             effort = effort_estimates.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L752_50", 1.0))
12924:             impact = impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L753_47", 0

```

```

.5))
12925:
12926:     # Calculate ratio (avoid division by zero)
12927:     ratio = impact / max(0.001, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L756_41", 0.1))
12928:
12929:     prioritized.append({
12930:         **gap,
12931:         "priority_ratio": ratio,
12932:         "effort": effort,
12933:         "impact": impact
12934:     })
12935:
12936:     # Sort by priority ratio (descending)
12937:     prioritized.sort(key=lambda x: x["priority_ratio"], reverse=True)
12938:
12939:     return prioritized
12940:
12941: def _assign_phases(
12942:     self,
12943:     prioritized_gaps: list[dict[str, Any]],
12944:     dependency_graph: dict[str, list[str]],
12945:     effort_estimates: dict[str, float]
12946: ) -> list[dict[str, Any]]:
12947:     """Assign gaps to time phases respecting dependencies"""
12948:     phases = [
12949:         {"name": "0-3m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L778_54", 0.0), "max_effort": 9.0},
12950:         {"name": "3-6m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L779_54", 0.0), "max_effort": 9.0},
12951:         {"name": "6-12m", "actions": [], "effort": ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L780_55", 0.0), "max_effort": 18.0}
12952:     ]
12953:
12954:     assigned = set()
12955:     gap_dict = {gap.get("id"): gap for gap in prioritized_gaps}
12956:
12957:     # Process gaps, but assign dependencies first
12958:     def assign_gap_recursive(gap_id: str, visited: set) -> None:
12959:         """Recursively assign gap and its dependencies"""
12960:         if gap_id in assigned or gap_id in visited:
12961:             return
12962:
12963:         visited.add(gap_id)
12964:
12965:         # First assign dependencies
12966:         dependencies = dependency_graph.get(gap_id, [])
12967:         for dep_id in dependencies:
12968:             if dep_id in gap_dict:
12969:                 assign_gap_recursive(dep_id, visited)
12970:
12971:         # Now assign this gap
12972:         if gap_id not in assigned and gap_id in gap_dict:
12973:             gap = gap_dict[gap_id]
12974:             effort = gap.get("effort", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L803_43", 1.0))
12975:
12976:             # Find earliest phase where all dependencies are satisfied

```

```

12977:             earliest_phase = self._get_earliest_phase(dependencies, assigned, phases)
12978:
12979:             # Assign to earliest phase with capacity
12980:             for i in range(earliest_phase, len(phases)):
12981:                 if phases[i]["effort"] + effort <= phases[i]["max_effort"]:
12982:                     phases[i]["actions"].append(gap)
12983:                     phases[i]["effort"] += effort
12984:                     assigned.add(gap_id)
12985:                     break
12986:
12987:             # Assign gaps in priority order, but respecting dependencies
12988:             for gap in prioritized_gaps:
12989:                 gap_id = gap.get("id", "unknown")
12990:                 assign_gap_recursive(gap_id, set())
12991:
12992:             return phases
12993:
12994:     def _get_earliest_phase(
12995:         self,
12996:         dependencies: list[str],
12997:         assigned: set,
12998:         phases: list[dict[str, Any]]
12999:     ) -> int:
13000:         """Get earliest phase where all dependencies are satisfied"""
13001:         if not dependencies:
13002:             return 0
13003:
13004:         max_dep_phase = -1
13005:         for dep_id in dependencies:
13006:             # Find which phase the dependency is in
13007:             dep_found = False
13008:             for i, phase in enumerate(phases):
13009:                 for action in phase["actions"]:
13010:                     if action.get("id") == dep_id:
13011:                         max_dep_phase = max(max_dep_phase, i)
13012:                         dep_found = True
13013:                         break
13014:             if dep_found:
13015:                 break
13016:
13017:         # Return phase after latest dependency (or 0 if no dependencies found yet)
13018:         return min(max_dep_phase + 1, len(phases) - 1) if max_dep_phase >= 0 else 0
13019:
13020:     def _calculate_total_uplift(
13021:         self,
13022:         phases: list[dict[str, Any]],
13023:         impact_scores: dict[str, float]
13024:     ) -> float:
13025:         """Calculate total expected uplift across all phases"""
13026:         total = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "total", 0.0) # Refactored
13027:
13028:         for phase in phases:
13029:             for action in phase["actions"]:
13030:                 gap_id = action.get("id", "unknown")
13031:                 impact = impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L860_51
", 0.0))

```

```
13032:             total += impact
13033:
13034:         return total
13035:
13036:     def _identify_critical_path(
13037:         self,
13038:         dependency_graph: dict[str, list[str]],
13039:         impact_scores: dict[str, float]
13040:     ) -> list[str]:
13041:         """Identify critical dependency chain"""
13042:         # Find the path with highest total impact
13043:         # Simple heuristic: find longest chain with high-impact nodes
13044:
13045:         # Find nodes with no dependents (endpoints)
13046:         has_dependents = set()
13047:         for deps in dependency_graph.values():
13048:             has_dependents.update(deps)
13049:
13050:         endpoints = [
13051:             gap_id for gap_id in dependency_graph
13052:             if gap_id not in has_dependents
13053:         ]
13054:
13055:         # For each endpoint, trace back to find highest-impact path
13056:         best_path = []
13057:         best_impact = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "best_impact", 0.0) # Refactored
13058:
13059:         for endpoint in endpoints:
13060:             path = self._trace_path(endpoint, dependency_graph)
13061:             path_impact = sum(impact_scores.get(gap_id, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L8
90_56", 0.0)) for gap_id in path)
13062:
13063:             if path_impact > best_impact:
13064:                 best_impact = path_impact
13065:                 best_path = path
13066:
13067:         return best_path
13068:
13069:     def _trace_path(
13070:         self,
13071:         gap_id: str,
13072:         dependency_graph: dict[str, list[str]]
13073:     ) -> list[str]:
13074:         """Trace dependency path from gap to root"""
13075:         path = [gap_id]
13076:         dependencies = dependency_graph.get(gap_id, [])
13077:
13078:         if dependencies:
13079:             # Follow first dependency (simplified)
13080:             dep_path = self._trace_path(dependencies[0], dependency_graph)
13081:             path = dep_path + path
13082:
13083:         return path
13084:
13085:     def _estimate_resources(
13086:         self,
```

```

13087:     phases: list[dict[str, Any]],
13088:     effort_estimates: dict[str, float]
13089: ) -> dict[str, Any]:
13090:     """Estimate resource requirements per phase"""
13091:     resources = {}
13092:
13093:     for phase in phases:
13094:         phase_name = phase["name"]
13095:         total_effort = phase["effort"]
13096:         num_actions = len(phase["actions"])
13097:
13098:         # Estimate team size (assuming 3 months per person-month per phase)
13099:         phase_months = {"0-3m": 3, "3-6m": 3, "6-12m": 6}
13100:         months = phase_months.get(phase_name, 3)
13101:         team_size = max(1, int(total_effort / months))
13102:
13103:         resources[phase_name] = {
13104:             "total_effort_months": total_effort,
13105:             "recommended_team_size": team_size,
13106:             "num_actions": num_actions
13107:         }
13108:
13109:     return resources
13110:
13111: # =====
13112: # MACRO PROMPT 5: PEER NORMALIZATION & CONFIDENCE SCALING
13113: # =====
13114:
13115: class PeerNormalizer:
13116:     """
13117:     ROLE: Macro Peer Evaluator [evaluation design]
13118:     GOAL: Ajustar clasificaciÃ³n macro considerando comparativos regionales.
13119:
13120:     INPUTS:
13121:     - convergence_by_policy_area
13122:     - peer_distributions: {policy_area -> {mean, std}}
13123:     - baseline_confidence
13124:
13125:     MANDATES:
13126:     - Calcular z-scores
13127:     - Penalizar si >k Ã¡reas estÃ;n < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L956_37", 1.0) z
13128:     - Aumentar confianza si todas dentro Ã±ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L957_42", 0.5)
z y dispersiÃ³n baja
13129:
13130:     OUTPUT:
13131:     JSON {z_scores, adjusted_confidence}
13132:
13133:     DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
13134:     Metadata:
13135:         - creation_date: 2024-01-15
13136:         - last_used_date: null
13137:         - intended_phase: future_phase_7_9
13138:         - deprecation_status: candidate_for_removal_2025Q2
13139:     """
13140:
13141:     def __init__(


```

```
13142:         self,
13143:         penalty_threshold: int = 3,
13144:         outlier_z_threshold: float = 2.0
13145:     ) -> None:
13146:     """
13147:     Initialize Peer Normalizer
13148:
13149:     Args:
13150:         penalty_threshold: Number of low-performing areas to trigger penalty
13151:         outlier_z_threshold: Z-score threshold for outlier identification
13152:     """
13153:     warnings.warn(
13154:         "PeerNormalizer is deprecated and scheduled for removal in 2025Q2. "
13155:         "No usage detected as of February 2025. Intended for future_phase_7_9. "
13156:         "Metadata: creation_date=2024-01-15, last_used_date=null, "
13157:         "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
13158:         DeprecationWarning,
13159:         stacklevel=2
13160:     )
13161:     self.k = penalty_threshold
13162:     self.outlier_z = outlier_z_threshold
13163:     logger.info(f"PeerNormalizer initialized (k={self.k}, z_outlier={self.outlier_z})")
13164:
13165:     def normalize(
13166:         self,
13167:         convergence_by_policy_area: dict[str, float],
13168:         peer_distributions: dict[str, dict[str, float]],
13169:         baseline_confidence: float
13170:     ) -> PeerNormalization:
13171:     """
13172:         Normalize scores against peer distributions
13173:
13174:         Args:
13175:             convergence_by_policy_area: Scores by policy area
13176:             peer_distributions: Mean and std dev for each policy area
13177:             baseline_confidence: Starting confidence level
13178:
13179:         Returns:
13180:             PeerNormalization with adjusted confidence
13181:     """
13182:     # Calculate z-scores
13183:     z_scores = self._calculate_z_scores(
13184:         convergence_by_policy_area,
13185:         peer_distributions
13186:     )
13187:
13188:     # Identify outliers
13189:     outlier_areas = self._identify_outliers(z_scores)
13190:
13191:     # Count low-performing areas
13192:     low_performers = [
13193:         area for area, z in z_scores.items()
13194:         if z < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1008_20", 1.0)
13195:     ]
13196:
13197:     # Adjust confidence
```

```
13198:         adjusted_confidence = self._adjust_confidence(
13199:             baseline_confidence,
13200:             z_scores,
13201:             low_performers
13202:         )
13203:
13204:     # Determine peer position
13205:     peer_position = self._determine_position(z_scores)
13206:
13207:     return PeerNormalization(
13208:         z_scores=z_scores,
13209:         adjusted_confidence=adjusted_confidence,
13210:         peer_position=peer_position,
13211:         outlier_areas=outlier_areas,
13212:         metadata={
13213:             "num_policy_areas": len(convergence_by_policy_area),
13214:             "low_performers": len(low_performers),
13215:             "penalty_threshold": self.k
13216:         }
13217:     )
13218:
13219:     def _calculate_z_scores(
13220:         self,
13221:         convergence: dict[str, float],
13222:         peer_distributions: dict[str, dict[str, float]]
13223:     ) -> dict[str, float]:
13224:         """Calculate z-scores for each policy area"""
13225:         z_scores = {}
13226:
13227:         for area, score in convergence.items():
13228:             if area in peer_distributions:
13229:                 peer = peer_distributions[area]
13230:                 mean = peer.get("mean", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1044_40", 0.5))
13231:                 std = peer.get("std", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1045_38", 0.1))
13232:
13233:                 # Calculate z-score
13234:                 z = (score - mean) / std if std > 0 else ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L
1048_57", 0.0)
13235:
13236:                 z_scores[area] = z
13237:
13238:         return z_scores
13239:
13240:     def _identify_outliers(
13241:         self,
13242:         z_scores: dict[str, float]
13243:     ) -> list[str]:
13244:         """Identify outlier policy areas"""
13245:         outliers = []
13246:
13247:         for area, z in z_scores.items():
13248:             if abs(z) > self.outlier_z:
13249:                 outliers.append(area)
13250:
13251:         return outliers
13252:
```

```
13253:     def _adjust_confidence(
13254:         self,
13255:         baseline: float,
13256:         z_scores: dict[str, float],
13257:         low_performers: list[str]
13258:     ) -> float:
13259:         """Adjust confidence based on peer comparison"""
13260:         adjusted = baseline
13261:
13262:         # Penalize if too many low performers
13263:         if len(low_performers) > self.k:
13264:             penalty = ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1078_22", 0.1) * (len(low_performers) - self.k)
13265:             adjusted *= (ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1079_25", 1.0) - min(penalty, ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1079_44", 0.5)))
13266:
13267:             # Check if all within ±ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1081_31", 0.5) z (tight distribution)
13268:             all_tight = all(abs(z) <= ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1082_34", 0.5) for z in z_scores.values())
13269:
13270:             if all_tight and len(z_scores) > 0:
13271:                 # Increase confidence for consistent performance
13272:                 adjusted *= 1.1
13273:
13274:             return max(ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1088_19", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1088_28", 1.0), adjusted))
13275:
13276:     def _determine_position(
13277:         self,
13278:         z_scores: dict[str, float]
13279:     ) -> str:
13280:         """Determine overall peer position"""
13281:         if not z_scores:
13282:             return "average"
13283:
13284:         avg_z = statistics.mean(z_scores.values())
13285:
13286:         if avg_z > ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1100_19", 0.5):
13287:             return "above_average"
13288:         elif avg_z < -ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.RoadmapOptimizer.__init__", "auto_param_L1102_22", 0.5):
13289:             return "below_average"
13290:         else:
13291:             return "average"
13292:
13293: # =====
13294: # MACRO PROMPTS FACADE
13295: # =====
13296:
13297: class MacroPromptsOrchestrator:
13298:     """
13299:         Orchestrator for all 5 macro-level analysis prompts
13300:
13301:         Provides unified interface to execute all macro analyses
13302:     """
13303:
```

```

13304:     def __init__(self) -> None:
13305:         """Initialize all macro prompt components"""
13306:         self.coverage_stressor = CoverageGapStressor()
13307:         self.contradiction_scanner = ContradictionScanner()
13308:         self.portfolio_composer = BayesianPortfolioComposer()
13309:         self.roadmap_optimizer = RoadmapOptimizer()
13310:         self.peer_normalizer = PeerNormalizer()
13311:
13312:         logger.info("MacroPromptsOrchestrator initialized with all 5 components")
13313:
13314:     def execute_all(
13315:         self,
13316:         macro_data: dict[str, Any]
13317:     ) -> dict[str, Any]:
13318:         """
13319:             Execute all 5 macro analyses
13320:
13321:             Args:
13322:                 macro_data: Complete macro-level data including:
13323:                     - convergence_by_dimension
13324:                     - convergence_by_policy_area
13325:                     - missing_clusters
13326:                     - dimension_coverage
13327:                     - policy_area_coverage
13328:                     - micro_claims
13329:                     - meso_summary_signals
13330:                     - macro_narratives
13331:                     - meso_posteriors
13332:                     - cluster_weights
13333:                     - critical_gaps
13334:                     - dependency_graph
13335:                     - effort_estimates
13336:                     - impact_scores
13337:                     - peer_distributions
13338:                     - baseline_confidence
13339:
13340:             Returns:
13341:                 Dict with results from all 5 analyses
13342:             """
13343:     results = {}
13344:
13345:     # 1. Coverage & Structural Gap Analysis
13346:     coverage_analysis = self.coverage_stressor.evaluate(
13347:         convergence_by_dimension=macro_data.get("convergence_by_dimension", {}),
13348:         missing_clusters=macro_data.get("missing_clusters", []),
13349:         dimension_coverage=macro_data.get("dimension_coverage", {}),
13350:         policy_area_coverage=macro_data.get("policy_area_coverage", {}),
13351:         baseline_confidence=macro_data.get("baseline_confidence", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.MacroPromptsOrchestrator.__init__", "auto_param_Ll165_70", 1.0))
13352:     )
13353:     results["coverage_analysis"] = asdict(coverage_analysis)
13354:
13355:     # 2. Inter-Level Contradiction Scan
13356:     contradiction_report = self.contradiction_scanner.scan(
13357:         micro_claims=macro_data.get("micro_claims", []),
13358:         meso_summary_signals=macro_data.get("meso_summary_signals", {}),

```

```

13359:         macro_narratives=macro_data.get("macro_narratives", {})
13360:     )
13361:     results["contradiction_report"] = asdict(contradiction_report)
13362:
13363:     # 3. Bayesian Portfolio Composition
13364:     bayesian_portfolio = self.portfolio_composer.compose(
13365:         meso_posteriors=macro_data.get("meso_posteriors", {}),
13366:         cluster_weights=macro_data.get("cluster_weights", {}),
13367:         reconciliation_penalties=macro_data.get("reconciliation_penalties")
13368:     )
13369:     results["bayesian_portfolio"] = asdict(bayesian_portfolio)
13370:
13371:     # 4. Roadmap Optimization
13372:     implementation_roadmap = self.roadmap_optimizer.optimize(
13373:         critical_gaps=macro_data.get("critical_gaps", []),
13374:         dependency_graph=macro_data.get("dependency_graph", {}),
13375:         effort_estimates=macro_data.get("effort_estimates", {}),
13376:         impact_scores=macro_data.get("impact_scores", {})
13377:     )
13378:     results["implementation_roadmap"] = asdict(implementation_roadmap)
13379:
13380:     # 5. Peer Normalization
13381:     peer_normalization = self.peer_normalizer.normalize(
13382:         convergence_by_policy_area=macro_data.get("convergence_by_policy_area", {}),
13383:         peer_distributions=macro_data.get("peer_distributions", {}),
13384:         baseline_confidence=macro_data.get("baseline_confidence", ParameterLoaderV2.get("farfan_core.analysis.macro_prompts.MacroPromptsOrchestrator._init__", "auto_param_L1198_70", 1.0))
13385:     )
13386:     results["peer_normalization"] = asdict(peer_normalization)
13387:
13388:     logger.info("Completed all 5 macro analyses")
13389:     return results
13390:
13391:
13392:
13393: =====
13394: FILE: src/farfan_pipeline/analysis/meso_cluster_analysis.py
13395: =====
13396:
13397: """Meso-level analytics utilities for cluster evaluation prompts.
13398:
13399: This module implements four independent helper functions that operationalise
13400: the bespoke "Prompt Meso" specifications used by the analytics team:
13401:
13402: * :func:`analyze_policy_dispersion` provides dispersion analytics, including
13403:   coefficient of variation, gap analysis, and a light penalty framework.
13404: * :func:`reconcile_cross_metrics` validates heterogeneous metric feeds against
13405:   an authoritative macro reference and emits governance flags.
13406: * :func:`compose_cluster_posterior` aggregates micro posteriors using a
13407:   Bayesian-style roll-up while accounting for reconciliation penalties.
13408: * :func:`calibrate_against_peers` situates the cluster against its peer group
13409:   using inter-quartile comparisons and Tukey-style outlier detection.
13410:
13411: The functions deliberately return both structured JSON-friendly payloads and a
13412: short narrative string whenever the prompt mandates qualitative guidance. The
13413: implementation is dependency-light (standard library only) to keep it aligned

```

```
13414: with the rest of the analytics toolbox.
13415: """
13416:
13417: from __future__ import annotations
13418:
13419: from dataclasses import dataclass
13420: from functools import reduce
13421: from statistics import fmean, pstdev
13422: from typing import TYPE_CHECKING
13423: from farfan_pipeline.core.parameters import ParameterLoaderV2
13424: from farfan_pipeline.core.calibration.decorators import calibrated_method
13425:
13426: if TYPE_CHECKING:
13427:     from collections.abc import Iterable, Mapping, Sequence
13428:
13429: def _to_float_sequence(values: Iterable[float]) -> list[float]:
13430:     return [float(v) for v in values]
13431:
13432: def _safe_mean(values: Iterable[float]) -> float:
13433:     seq = _to_float_sequence(values)
13434:     if not seq:
13435:         return 0.0
13436:     return float(fmean(seq))
13437:
13438: def _safe_std(values: Iterable[float]) -> float:
13439:     seq = _to_float_sequence(values)
13440:     if len(seq) <= 1:
13441:         return 0.0
13442:     return float(pstdev(seq))
13443:
13444: def _percentile(values: Sequence[float], percent: float) -> float:
13445:     seq = sorted(_to_float_sequence(values))
13446:     if not seq:
13447:         return 0.0
13448:     if percent <= 0:
13449:         return seq[0]
13450:     if percent >= 100:
13451:         return seq[-1]
13452:     k = (len(seq) - 1) * (percent / 100.0)
13453:     lower_index = int(k)
13454:     upper_index = min(lower_index + 1, len(seq) - 1)
13455:     weight = k - lower_index
13456:     return seq[lower_index] + weight * (seq[upper_index] - seq[lower_index])
13457:
13458: def _gini(values: Iterable[float]) -> float:
13459:     """Compute the Gini coefficient for a sequence of non-negative values."""
13460:
13461:     seq = sorted(_to_float_sequence(values))
13462:     if not seq:
13463:         return 0.0
13464:     if any(v < 0 for v in seq):
13465:         raise ValueError("Gini coefficient is undefined for negative values")
13466:     if all(v == 0 for v in seq):
13467:         return 0.0
13468:     total = sum(seq)
13469:     if total == 0:
```

```
13470:     # Non-negative numbers can only sum to zero if they are all zero, which is
13471:     # handled above. Guard against floating point artefacts that leave a
13472:     # near-zero denominator.
13473:     return 0.0
13474: n = len(seq)
13475: weighted_sum = 0.0
13476: for i, value in enumerate(seq, start=1):
13477:     weighted_sum += i * value
13478: gini = (2 * weighted_sum) / (n * total) - (n + 1) / n
13479: return float(gini)
13480:
13481: def _tukey_bounds(p25: float, p75: float) -> tuple[float, float]:
13482:     lower_quartile, upper_quartile = sorted((float(p25), float(p75)))
13483:     iqr = upper_quartile - lower_quartile
13484:     return (lower_quartile - 1.5 * iqr, upper_quartile + 1.5 * iqr)
13485:
13486: def analyze_policy_dispersion(
13487:     policy_area_scores: Mapping[str, float],
13488:     peer_dispersion_stats: Mapping[str, float],
13489:     thresholds: Mapping[str, float],
13490: ) -> tuple[dict[str, object], str]:
13491:     """Evaluate intra-cluster dispersion and recommend a penalty.
13492:
13493:     Parameters
13494:     -----
13495:     policy_area_scores:
13496:         Mapping of policy area names to their normalised scores.
13497:     peer_dispersion_stats:
13498:         Median dispersion statistics for comparable clusters. Expected keys are
13499:         ``cv_median`` and ``gap_median``; missing keys are handled gracefully.
13500:     thresholds:
13501:         Warning/failure thresholds with keys ``cv_warn``, ``cv_fail``,
13502:         ``gap_warn`` and ``gap_fail``.
13503:
13504:     Returns
13505:     -----
13506:     Tuple[Dict[str, object], str]
13507:         A tuple of the JSON-friendly payload and the five-to-six line narrative.
13508: """
13509:
13510:     values = _to_float_sequence(policy_area_scores.values())
13511:     mean_score = _safe_mean(values)
13512:     std_score = _safe_std(values)
13513:     cv = std_score / mean_score if mean_score else 0.0
13514:     max_gap = float(max(values) - min(values)) if values else 0.0
13515:     gini = _gini(values)
13516:
13517:     peer_cv = float(peer_dispersion_stats.get("cv_median", cv))
13518:     peer_gap = float(peer_dispersion_stats.get("gap_median", max_gap))
13519:
13520:     cv_warn = float(thresholds.get("cv_warn", peer_cv))
13521:     cv_fail = float(thresholds.get("cv_fail", peer_cv))
13522:     gap_warn = float(thresholds.get("gap_warn", peer_gap))
13523:     gap_fail = float(thresholds.get("gap_fail", peer_gap))
13524:
13525:     severity = 0
```

```
13526:     if cv > cv_warn or max_gap > gap_warn:
13527:         severity = 1
13528:     if cv > cv_fail or max_gap > gap_fail:
13529:         severity = 2
13530:     peerEscalation = cv > 1.5 * peer_cv or max_gap > 1.5 * peer_gap
13531:     if peerEscalation or cv > 1.5 * cv_fail or max_gap > 1.5 * gap_fail:
13532:         severity = 3
13533:
13534:     classification = {
13535:         0: "Concentrado",
13536:         1: "Moderado",
13537:         2: "Disperso",
13538:         3: "CrÃ-tico",
13539:     }[severity]
13540:
13541:     penaltyComponents: list[float] = []
13542:     if cv_fail:
13543:         penaltyComponents.append(min(cv / cv_fail, 1.5))
13544:     if gap_fail:
13545:         penaltyComponents.append(min(max_gap / gap_fail, 1.5))
13546:     peerSignal: list[float] = []
13547:     if peer_cv:
13548:         peerSignal.append(min(cv / peer_cv, 2.0))
13549:     if peer_gap:
13550:         peerSignal.append(min(max_gap / peer_gap, 2.0))
13551:
13552:     basePenalty = _safe_mean(penaltyComponents) if penaltyComponents else 0.0
13553:     peerPenalty = _safe_mean(peerSignal) if peerSignal else 0.0
13554:     penalty = float(min(1.0, 0.6 * basePenalty + 0.4 * (peerPenalty - 1.0)))
13555:     penalty = max(0.0, penalty)
13556:
13557: # Hypothetical normalisation of the lower tail: lift scores below Q1 to Q1.
13558:     if values:
13559:         q1 = float(_percentile(values, 25))
13560:         normalisedValues = [max(v, q1) for v in values]
13561:         normMean = _safe_mean(normalisedValues)
13562:         normCv = _safe_std(normalisedValues) / normMean if normMean else 0.0
13563:         normGap = float(max(normalisedValues) - min(normalisedValues))
13564:         meanUplift = normMean - meanScore
13565:     else:
13566:         normCv = 0.0
13567:         normGap = 0.0
13568:         meanUplift = 0.0
13569:
13570:     jsonPayload = {
13571:         "cv": cv,
13572:         "max_gap": max_gap,
13573:         "gini": gini,
13574:         "class": classification,
13575:         "penalty": penalty,
13576:         "normalized_projection": {
13577:             "adjusted_cv": normCv,
13578:             "adjusted_max_gap": normGap,
13579:             "mean_uplift": meanUplift,
13580:         },
13581:     }
```

```
13582:
13583:     lines = [
13584:         f"La variabilidad intracluster muestra un CV de {cv:.2f} frente al referente de {peer_cv:.2f}.",
13585:         f"La brecha máxima es de {max_gap:.1f} puntos, lo que sitúa la clasificación en nivel {classification}.",
13586:         f"El coeficiente de Gini ({gini:.2f}) evidencia {'alta' if gini > 0.3 else 'moderada'} concentración de resultados.",
13587:         "La penalización propuesta \210 pondera desalineaciones internas y diferenciales contra los pares comparables.",
13588:         "Si se normaliza la cola baja hacia el cuartil 25, el CV se reduce a "
13589:         f"{norm_cv:.2f} y el gap a {norm_gap:.1f} con un uplift medio de {mean_uplift:.1f}.",
13590:         "Persisten riesgos de sesgo de apreciación si se ignora la sensibilidad de la cola baja frente a shocks sectoriales.",
13591:     ]
13592:     narrative = "\n".join(lines[:6])
13593:
13594:     return json_payload, narrative
13595:
13596: @dataclass
13597: class MetricViolation:
13598:     metric_id: str
13599:     unit_mismatch: bool = False
13600:     stale_period: bool = False
13601:     entity_misalignment: bool = False
13602:     out_of_range: bool = False
13603:
13604:     @calibrated_method("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict")
13605:     def to_flag_dict(self) -> dict[str, object]:
13606:         return {
13607:             "metric_id": self.metric_id,
13608:             "unit_mismatch": self.unit_mismatch,
13609:             "stale_period": self.stale_period,
13610:             "entity_misalignment": self.entity_misalignment,
13611:             "out_of_range": self.out_of_range,
13612:         }
13613:
13614:     def _convert_unit(
13615:         value: float,
13616:         from_unit: str,
13617:         to_unit: str,
13618:         crosswalk: Mapping[str, Mapping[str, float]],
13619:     ) -> tuple[float, str]:
13620:         if from_unit == to_unit:
13621:             return value, to_unit
13622:         conversions = crosswalk.get(from_unit, {})
13623:         factor = conversions.get(to_unit)
13624:         if factor is None:
13625:             raise ValueError("Units are not convertible with provided crosswalk")
13626:         return value * factor, to_unit
13627:
13628:     def reconcile_cross_metrics(
13629:         aggregated_metrics: Iterable[Mapping[str, object]],
13630:         macro_json: Mapping[str, object],
13631:     ) -> dict[str, object]:
13632:         """Validate heterogeneous metrics against an authoritative macro source."""
13633:
13634:         reference: Mapping[str, Mapping[str, object]] = macro_json.get("metrics", {}) # type: ignore[assignment]
13635:         crosswalk: Mapping[str, Mapping[str, float]] = macro_json.get("unit_crosswalk", {}) # type: ignore[assignment]
13636:
13637:         validated_metrics: list[dict[str, object]] = []
```

```
13638:     violations: list[dict[str, object]] = []
13639:
13640:     for metric in aggregated_metrics:
13641:         metric_id = str(metric.get("metric_id"))
13642:         value = float(metric.get("value", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L246_42", 0.0)))
13643:         unit = str(metric.get("unit")) if metric.get("unit") is not None else ""
13644:         period = str(metric.get("period")) if metric.get("period") is not None else ""
13645:         entity = str(metric.get("entity")) if metric.get("entity") is not None else ""
13646:
13647:         expected = reference.get(metric_id, {})
13648:         expected_unit = str(expected.get("unit", unit)) if expected else unit
13649:         expected_period = str(expected.get("period", period)) if expected else period
13650:         expected_entities = expected.get("entities") if isinstance(expected.get("entities"), list) else []
13651:         lower_bound, upper_bound = expected.get("range", (None, None))
13652:
13653:         violation = MetricViolation(metric_id)
13654:
13655:         reconciled_value = value
13656:         reconciled_unit = unit
13657:
13658:         conversion_failed = False
13659:
13660:         if expected_unit and unit and unit != expected_unit:
13661:             try:
13662:                 reconciled_value, reconciled_unit = _convert_unit(value, unit, expected_unit, crosswalk)
13663:             except ValueError:
13664:                 violation.unit_mismatch = True
13665:                 conversion_failed = True
13666:
13667:         if expected_period and period and period != expected_period:
13668:             violation.stale_period = True
13669:
13670:         if expected_entities and entity and entity not in expected_entities:
13671:             violation.entity_misalignment = True
13672:
13673:         if not conversion_failed and lower_bound is not None and reconciled_value < float(lower_bound):
13674:             violation.out_of_range = True
13675:         if not conversion_failed and upper_bound is not None and reconciled_value > float(upper_bound):
13676:             violation.out_of_range = True
13677:
13678:         validated_metrics.append(
13679:             {
13680:                 "metric_id": metric_id,
13681:                 "value": reconciled_value,
13682:                 "unit": reconciled_unit,
13683:                 "period": expected_period if expected_period else period,
13684:                 "entity": entity,
13685:             }
13686:         )
13687:
13688:         if (
13689:             violation.unit_mismatch
13690:             or violation.stale_period
13691:             or violation.entity_misalignment
13692:             or violation.out_of_range
```

```

13693:         ) :
13694:             violations.append(violation.to_flag_dict())
13695:
13696:     total_checks = len(validated_metrics) * 4 if validated_metrics else 1
13697:     total_violations = sum(
13698:         violation[flag]
13699:         for violation in violations
13700:             for flag in ("unit_mismatch", "stale_period", "entity_misalignment", "out_of_range")
13701:     )
13702:     reconciled_confidence = max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L306_32", 0.0),
ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L306_37", 1.0) - total_violations / total_checks)
13703:
13704:     return {
13705:         "metrics_validated": validated_metrics,
13706:         "violations": violations,
13707:         "reconciled_confidence": reconciled_confidence,
13708:     }
13709:
13710: def compose_cluster_posterior(
13711:     micro_posteriors: Iterable[float],
13712:     weighting_trace: Iterable[float] | None = None,
13713:     reconciliation_penalties: Mapping[str, float] | None = None,
13714: ) -> tuple[dict[str, object], str]:
13715:     """Combine micro posteriors and reconciliation penalties into a cluster view."""
13716:
13717:     posts = _to_float_sequence(micro_posteriors)
13718:     if not posts:
13719:         raise ValueError("micro_posteriors cannot be empty")
13720:
13721:     if weighting_trace is None:
13722:         weights = [ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L326_19", 1.0)] * len(posts)
13723:     else:
13724:         weights = _to_float_sequence(weighting_trace)
13725:         if len(weights) != len(posts):
13726:             raise ValueError("weighting_trace must match micro_posteriors length")
13727:             if any(w < 0 for w in weights):
13728:                 raise ValueError("weighting_trace values must be non-negative")
13729:             if all(w == 0 for w in weights):
13730:                 weights = [ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L334_19", 1.0)] * len(posts)
13731:
13732:     # Prevent degenerate/negative totals; fallback to uniform if needed.
13733:     weights = [max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L337_19", 0.0), float(w)) fo
r w in weights]
13734:     total_weight = sum(weights)
13735:     if total_weight == 0:
13736:         raise ValueError("At least one weight must be positive")
13737:     normalised_weights = [w / total_weight for w in weights]
13738:     prior_meso = float(sum(p * w for p, w in zip(posts, normalised_weights, strict=True)))
13739:
13740:     variance = float(sum(w * (p - prior_meso) ** 2 for p, w in zip(posts, normalised_weights, strict=True)))
13741:     uncertainty_index = float(variance ** ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L345_
42", 0.5))
13742:
13743:     penalties_input = reconciliation_penalties or {}
13744:     dispersion_penalty = float(penalties_input.get("dispersion_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.t
o_flag_dict", "auto_param_L348_73", 0.0)))

```

```

13745:     coverage_penalty = float(penalties_input.get("coverage_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L349_69", 0.0)))
13746:     reconciliation_penalty = float(penalties_input.get("reconciliation_penalty", ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L350_81", 0.0)))
13747:
13748:     penalty_factor = reduce(
13749:         lambda acc, val: acc * val,
13750:         [
13751:             max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L355_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L355_21", 1.0) - dispersion_penalty),
13752:             max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L356_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L356_21", 1.0) - coverage_penalty),
13753:             max(ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L357_16", 0.0), ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L357_21", 1.0) - reconciliation_penalty),
13754:         ],
13755:         ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L359_8", 1.0),
13756:     )
13757:     posterior_meso = float(prior_meso * penalty_factor)
13758:
13759:     json_payload = {
13760:         "prior_meso": prior_meso,
13761:         "penalties": {
13762:             "dispersion_penalty": dispersion_penalty,
13763:             "coverage_penalty": coverage_penalty,
13764:             "reconciliation_penalty": reconciliation_penalty,
13765:         },
13766:         "posterior_meso": posterior_meso,
13767:         "uncertainty_index": uncertainty_index,
13768:     }
13769:
13770:     explanation_lines = [
13771:         f"La media ponderada de las micro evidencias define un prior meso de {prior_meso:.3f}.",
13772:         f"Las penalizaciones por dispersiÃ³n, cobertura y reconciliaciÃ³n actÃºan de forma multiplicativa sobre el prior.",
13773:         f"El ajuste integrado produce un posterior de {posterior_meso:.3f}, coherente con la gobernanza aplicada.",
13774:         f"La incertidumbre residual ({\u00d7}203 \u2248211\u2248210 {uncertainty_index:.3f}) refleja la varianza remanente de las micro posteriors.",
13775:     ]
13776:
13777:     return json_payload, "\n".join(explanation_lines)
13778:
13779: def calibrate_against_peers(
13780:     policy_area_scores: Mapping[str, float],
13781:     peer_context: Mapping[str, Mapping[str, float]],
13782: ) -> tuple[dict[str, object], str]:
13783:     """Compare cluster scores against peer medians and inter-quartile ranges."""
13784:
13785:     area_positions: dict[str, str] = {}
13786:     outliers: dict[str, bool] = {}
13787:     dispersion_values = _to_float_sequence(policy_area_scores.values())
13788:     if dispersion_values:
13789:         cluster_mean = _safe_mean(dispersion_values)
13790:         cluster_std = _safe_std(dispersion_values)
13791:         cluster_cv = cluster_std / cluster_mean if cluster_mean else ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "auto_param_L395_69", 0.0)
13792:     else:
13793:         cluster_cv = ParameterLoaderV2.get("farfan_core.analysis.meso_cluster_analysis.MetricViolation.to_flag_dict", "cluster_cv", 0.0) # Refactored
13794:

```

```

13795:     for area, score in policy_area_scores.items():
13796:         peers = peer_context.get(area, {})
13797:         median = float(peers.get("median", score))
13798:         p25 = float(peers.get("p25", median))
13799:         p75 = float(peers.get("p75", median))
13800:         p75 - p25
13801:
13802:         if score < p25:
13803:             area_positions[area] = "below"
13804:         elif score > p75:
13805:             area_positions[area] = "above"
13806:         else:
13807:             area_positions[area] = "within"
13808:
13809:         lower_bound, upper_bound = _tukey_bounds(p25, p75)
13810:         outliers[area] = score < lower_bound or score > upper_bound
13811:
13812:     json_payload = {
13813:         "area_positions": area_positions,
13814:         "outliers": outliers,
13815:     }
13816:
13817:     above_areas = [area for area, position in area_positions.items() if position == "above"]
13818:     below_areas = [area for area, position in area_positions.items() if position == "below"]
13819:     within_areas = [area for area, position in area_positions.items() if position == "within"]
13820:
13821:     narrative_lines = [
13822:         "El contraste con la mediana de los pares muestra un desempeño heterogéneo por área." ,
13823:         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 casas relevantes'} quedan por debajo.", 
13824:         f"Las áreas en zona intercuartílica ({', '.join(within_areas) if within_areas else 'sin registros'}) sostienen la base del clúster.", 
13825:         "Los outliers detectados mediante Tukey advierten focos críticos que requieren revisión específica.", 
13826:         f"Un municipio con media equiparable pero menor CV (~{cluster_cv:.2f}) ofrecerá una narrativa más cohesionada, subrayando nuestra dispersión relativa.", 
13827:         "Conviene integrar estos hallazgos en la calibración narrativa para evitar sobreponderar éxitos aislados frente a rezagos estructurales.", 
13828:         "Recomendar explicitar cómo la dispersión condiciona la comparabilidad con pares que exhiben mayor equilibrio interno.", 
13829:     ]
13830:
13831:     return json_payload, "\n".join(narrative_lines[:7])
13832:
13833:
13834:
13835:
13836: =====
13837: FILE: src/farfán_pipeline/analysis/micro_prompts.py
13838: =====
13839:
13840: """
13841: Micro Prompts - Provenance Auditor, Bayesian Posterior Justification, and Anti-Milagro Stress Test
13842: =====
13843:
13844: This module implements three critical micro-level analysis prompts:
13845:
13846: 1. PROVENANCE AUDITOR (QMCM Integrity Check):
13847:     - Validates Questionnaire\206\222Method Contribution Map consistency
13848:     - Verifies provenance DAG integrity

```

```
13849:     - Detects orphan nodes and schema mismatches
13850:     - Monitors timing anomalies
13851:
13852: 2. BAYESIAN POSTERIOR JUSTIFICATION:
13853:     - Explains signal contributions to posterior probability
13854:     - Ranks signals by marginal impact
13855:     - Identifies discarded signals
13856:     - Justifies test types (Hoop, Smoking-Gun, etc.)
13857:
13858: 3. ANTI-MILAGRO STRESS TEST:
13859:     - Detects structural fragility in causal chains
13860:     - Evaluates proportionality pattern density
13861:     - Simulates node removal to test robustness
13862:     - Identifies non-proportional jumps
13863:
13864: Author: Integration Team
13865: Version: 1.0.0
13866: Python: 3.10+
13867: """
13868:
13869: from __future__ import annotations
13870:
13871: import logging
13872: import time
13873: import warnings
13874: from collections import defaultdict
13875: from dataclasses import asdict, dataclass, field
13876: from typing import Any
13877:
13878: import numpy as np
13879:
13880: # from farfan_pipeline import get_parameter_loader # CALIBRATION DISABLED
13881: from farfan_pipeline.core.calibration.decorators import calibrated_method
13882:
13883: logger = logging.getLogger(__name__)
13884:
13885: # =====
13886: # PROVENANCE AUDITOR - QMCM INTEGRITY CHECK
13887: # =====
13888:
13889: @dataclass
13890: class QMCMRecord:
13891:     """Record in the Questionnaire\206\222Method Contribution Map
13892:
13893:     Aligned with questionnaire_monolith.json structure:
13894:     - base_slot: Question slot identifier from monolith
13895:     - scoring_modality: Scoring mechanism (binary, ordinal, numeric, etc.)
13896:     """
13897:     question_id: str
13898:     method_fqn: str
13899:     contribution_weight: float
13900:     timestamp: float
13901:     output_schema: dict[str, Any]
13902:     base_slot: str | None = field(default=None) # From questionnaire monolith
13903:     scoring_modality: str | None = field(default=None) # From questionnaire monolith
13904:     metadata: dict[str, Any] = field(default_factory=dict)
```

```
13905:  
13906: @dataclass  
13907: class ProvenanceNode:  
13908:     """Node in the provenance DAG"""  
13909:     node_id: str  
13910:     node_type: str # 'input', 'method', 'output'  
13911:     parent_ids: list[str]  
13912:     qmcm_record_id: str | None = None  
13913:     timing: float = 0.0  
13914:     metadata: dict[str, Any] = field(default_factory=dict)  
13915:  
13916: @dataclass  
13917: class ProvenanceDAG:  
13918:     """Provenance directed acyclic graph"""  
13919:     nodes: dict[str, ProvenanceNode]  
13920:     edges: list[tuple[str, str]] # (from_node_id, to_node_id)  
13921:  
13922:     @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceDAG.get_root_nodes")  
13923:     def get_root_nodes(self) -> list[str]:  
13924:         """Get nodes without parents (primary inputs)"""  
13925:         return [nid for nid, node in self.nodes.items() if not node.parent_ids]  
13926:  
13927:     @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceDAG.get_orphan_nodes")  
13928:     def get_orphan_nodes(self) -> list[str]:  
13929:         """Get nodes without parents that are not primary inputs"""  
13930:         return [  
13931:             nid for nid, node in self.nodes.items()  
13932:             if not node.parent_ids and node.node_type != 'input'  
13933:         ]  
13934:  
13935: @dataclass  
13936: class AuditResult:  
13937:     """Result of provenance audit"""  
13938:     missing_qmcm: list[str] # Node IDs without QMCM records  
13939:     orphan_nodes: list[str] # Nodes without proper parents  
13940:     schema_mismatches: list[dict[str, Any]] # Schema violations  
13941:     latency_anomalies: list[dict[str, Any]] # Timing outliers  
13942:     contribution_weights: dict[str, float] # Method contribution distribution  
13943:     severity: str # 'LOW', 'MEDIUM', 'HIGH', 'CRITICAL'  
13944:     narrative: str # 3-4 line explanation  
13945:     timestamp: float = field(default_factory=time.time)  
13946:  
13947: class ProvenanceAuditor:  
13948:     """  
13949:     ROLE: Provenance Auditor [data governance]  
13950:     GOAL: Verify QMCM consistency and provenance DAG integrity  
13951:  
13952:     DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.  
13953:     Metadata:  
13954:         - creation_date: 2024-01-15  
13955:         - last_used_date: null  
13956:         - intended_phase: future_phase_7_9  
13957:         - deprecation_status: candidate_for_removal_2025Q2  
13958:     """  
13959:  
13960:     def __init__(
```

```

13961:         self,
13962:         p95_latency_threshold: float | None = None,
13963:         method_contracts: dict[str, dict[str, Any]] | None = None
13964:     ) -> None:
13965:         """
13966:             Initialize provenance auditor
13967:
13968:             Args:
13969:                 p95_latency_threshold: Historical p95 latency for anomaly detection
13970:                 method_contracts: Expected output schemas by method
13971:
13972:             """
13973:             warnings.warn(
13974:                 "ProvenanceAuditor is deprecated and scheduled for removal in 2025Q2. "
13975:                 "No usage detected as of February 2025. Intended for future_phase_7_9. "
13976:                 "Metadata: creation_date=2024-01-15, last_used_date=null, "
13977:                 "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
13978:                 DeprecationWarning,
13979:                 stacklevel=2
13980:             )
13981:             self.p95_threshold = p95_latency_threshold or 1000.0
13982:             self.method_contracts = method_contracts or {}
13983:             self.logger = logging.getLogger(self.__class__.__name__)
13984:
13985:     def audit(
13986:         self,
13987:         micro_answer: Any, # MicroLevelAnswer object
13988:         evidence_registry: dict[str, QMCMRecord],
13989:         provenance_dag: ProvenanceDAG,
13990:         method_contracts: dict[str, dict[str, Any]] | None = None
13991:     ) -> AuditResult:
13992:
13993:         """
13994:             Perform comprehensive provenance audit
13995:
13996:             MANDATES:
13997:                 1. Validate 1:1 correspondence between DAG nodes and QMCM records
13998:                 2. Confirm no orphan nodes (except primary inputs)
13999:                 3. Check timing drift (flag if > p95 historical)
14000:                 4. Verify output_schema compliance
14001:                 5. Emit JSON audit + narrative
14002:
14003:             Args:
14004:                 micro_answer: MicroLevelAnswer object to audit
14005:                 evidence_registry: QMCM records indexed by ID
14006:                 provenance_dag: Provenance DAG structure
14007:                 method_contracts: Expected schemas (optional override)
14008:
14009:             Returns:
14010:                 AuditResult with findings and severity assessment
14011:
14012:                 contracts = method_contracts or self.method_contracts
14013:
14014:                 # 1. Validate QMCM correspondence
14015:                 missing_qmcm = self._check_qmcm_correspondence(provenance_dag, evidence_registry)
14016:
14017:                 # 2. Detect orphan nodes
14018:                 orphan_nodes = provenance_dag.get_orphan_nodes()

```

```
14017:  
14018:      # 3. Check timing anomalies  
14019:      latency_anomalies = self._check_latency_anomalies(provenance_dag)  
14020:  
14021:      # 4. Verify schema compliance  
14022:      schema_mismatches = self._check_schema_compliance(  
14023:          provenance_dag, evidence_registry, contracts  
14024:      )  
14025:  
14026:      # 5. Calculate contribution weights  
14027:      contribution_weights = self._calculate_contribution_weights(evidence_registry)  
14028:  
14029:      # Determine severity  
14030:      severity = self._assess_severity(  
14031:          missing_qmcm, orphan_nodes, schema_mismatches, latency_anomalies  
14032:      )  
14033:  
14034:      # Generate narrative  
14035:      narrative = self._generate_narrative(  
14036:          len(missing_qmcm), len(orphan_nodes),  
14037:          len(schema_mismatches), len(latency_anomalies), severity  
14038:      )  
14039:  
14040:      return AuditResult(  
14041:          missing_qmcm=missing_qmcm,  
14042:          orphan_nodes=orphan_nodes,  
14043:          schema_mismatches=schema_mismatches,  
14044:          latency_anomalies=latency_anomalies,  
14045:          contribution_weights=contribution_weights,  
14046:          severity=severity,  
14047:          narrative=narrative  
14048:      )  
14049:  
14050:      def _check_qmcm_correspondence(  
14051:          self, dag: ProvenanceDAG, registry: dict[str, QMCMRecord]  
14052:      ) -> list[str]:  
14053:          """Check 1:1 node-to-QMCM correspondence"""  
14054:          missing = []  
14055:          for node_id, node in dag.nodes.items():  
14056:              if node.node_type == 'method':  
14057:                  if not node.qmcm_record_id or node.qmcm_record_id not in registry:  
14058:                      missing.append(node_id)  
14059:          return missing  
14060:  
14061:      @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor._check_latency_anomalies")  
14062:      def _check_latency_anomalies(self, dag: ProvenanceDAG) -> list[dict[str, Any]]:  
14063:          """Detect timing outliers beyond p95 threshold"""  
14064:          anomalies = []  
14065:          for node_id, node in dag.nodes.items():  
14066:              if node.timing > self.p95_threshold:  
14067:                  anomalies.append({  
14068:                      'node_id': node_id,  
14069:                      'timing': node.timing,  
14070:                      'threshold': self.p95_threshold,  
14071:                      'excess': node.timing - self.p95_threshold  
14072:                  })
```

```
14073:         return anomalies
14074:
14075:     def _check_schema_compliance(
14076:         self,
14077:         dag: ProvenanceDAG,
14078:         registry: dict[str, QMCMRecord],
14079:         contracts: dict[str, dict[str, Any]]
14080:     ) -> list[dict[str, Any]]:
14081:         """Verify method outputs match expected schemas"""
14082:         mismatches = []
14083:         for node_id, node in dag.nodes.items():
14084:             if node.node_type == 'method' and node.qmcm_record_id:
14085:                 record = registry.get(node.qmcm_record_id)
14086:                 if record and record.method_fqn in contracts:
14087:                     expected = contracts[record.method_fqn]
14088:                     actual = record.output_schema
14089:
14090:                     if not self._schemas_match(expected, actual):
14091:                         mismatches.append({
14092:                             'node_id': node_id,
14093:                             'method': record.method_fqn,
14094:                             'expected_schema': expected,
14095:                             'actual_schema': actual
14096:                         })
14097:         return mismatches
14098:
14099:     @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor._schemas_match")
14100:     def _schemas_match(self, expected: dict[str, Any], actual: dict[str, Any]) -> bool:
14101:         """Check if actual schema matches expected schema"""
14102:         # Simple type-based matching
14103:         return all(key in actual for key, expected_type in expected.items())
14104:
14105:     def _calculate_contribution_weights(
14106:         self, registry: dict[str, QMCMRecord]
14107:     ) -> dict[str, float]:
14108:         """Calculate method contribution distribution"""
14109:         weights = defaultdict(float)
14110:         for record in registry.values():
14111:             weights[record.method_fqn] += record.contribution_weight
14112:         return dict(weights)
14113:
14114:     def _assess_severity(
14115:         self,
14116:         missing_qmcm: list[str],
14117:         orphan_nodes: list[str],
14118:         schema_mismatches: list[dict[str, Any]],
14119:         latency_anomalies: list[dict[str, Any]]
14120:     ) -> str:
14121:         """Assess overall audit severity"""
14122:         total_issues = (
14123:             len(missing_qmcm) + len(orphan_nodes) +
14124:             len(schema_mismatches) + len(latency_anomalies)
14125:         )
14126:
14127:         if total_issues == 0:
14128:             return 'LOW'
```

```
14129:         elif total_issues <= 2:
14130:             return 'MEDIUM'
14131:         elif total_issues <= 5:
14132:             return 'HIGH'
14133:         else:
14134:             return 'CRITICAL'
14135:
14136:     def _generate_narrative(
14137:         self, missing: int, orphans: int, mismatches: int, anomalies: int, severity: str
14138:     ) -> str:
14139:         """Generate 3-4 line narrative summary"""
14140:         narrative = f"Provenance audit completed with {severity} severity. "
14141:
14142:         if missing > 0:
14143:             narrative += f"Found {missing} nodes without QMCM records. "
14144:         if orphans > 0:
14145:             narrative += f"Detected {orphans} orphan nodes requiring parent linkage. "
14146:         if mismatches > 0:
14147:             narrative += f"Identified {mismatches} schema violations. "
14148:         if anomalies > 0:
14149:             narrative += f"Flagged {anomalies} latency anomalies exceeding p95. "
14150:
14151:         if severity == 'LOW':
14152:             narrative += "All critical integrity checks passed."
14153:         elif severity == 'CRITICAL':
14154:             narrative += "Immediate remediation required for data governance."
14155:
14156:     return narrative
14157:
14158:     @calibrated_method("farfan_core.analysis.micro_prompts.ProvenanceAuditor.to_json")
14159:     def to_json(self, result: AuditResult) -> dict[str, Any]:
14160:         """Export audit result as JSON"""
14161:         return asdict(result)
14162:
14163: # =====
14164: # BAYESIAN POSTERIOR JUSTIFICATION
14165: # =====
14166:
14167: @dataclass
14168: class Signal:
14169:     """Signal contributing to posterior probability"""
14170:     test_type: str # 'Hoop', 'Smoking-Gun', 'Straw-in-Wind', 'Doubly-Decisive'
14171:     likelihood: float
14172:     weight: float
14173:     raw_evidence_id: str
14174:     reconciled: bool
14175:     delta_posterior: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.ProvenanceAuditor.to_json", "auto_param_L318_29", 0.0)
14176:     reason: str = ""
14177:
14178: @dataclass
14179: class PosteriorJustification:
14180:     """Bayesian posterior justification result"""
14181:     prior: float
14182:     posterior: float
14183:     signals_ranked: list[dict[str, Any]] # Signals sorted by |f\224|
14184:     discarded_signals: list[dict[str, Any]] # Signals rejected
```

```
14185:     anti_miracle_cap_applied: bool
14186:     cap_delta: float # How much was capped
14187:     robustness_narrative: str # 5-6 line synthesis
14188:     timestamp: float = field(default_factory=time.time)
14189:
14190: class BayesianPosteriorExplainer:
14191:     """
14192:         ROLE: Probabilistic Explainer [causal inference]
14193:         GOAL: Explain signal contributions to final posterior
14194:
14195:         DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
14196:         Metadata:
14197:             - creation_date: 2024-01-15
14198:             - last_used_date: null
14199:             - intended_phase: future_phase_7_9
14200:             - deprecation_status: candidate_for_removal_2025Q2
14201:         """
14202:
14203:     def __init__(self, anti_miracle_cap: float = 0.95) -> None:
14204:         """
14205:             Initialize Bayesian posterior explainer
14206:
14207:             Args:
14208:                 anti_miracle_cap: Maximum posterior probability (anti-miracle constraint)
14209:             """
14210:         warnings.warn(
14211:             "BayesianPosteriorExplainer is deprecated and scheduled for removal in 2025Q2. "
14212:             "No usage detected as of February 2025. Intended for future_phase_7_9. "
14213:             "Metadata: creation_date=2024-01-15, last_used_date=null, "
14214:             "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
14215:             DeprecationWarning,
14216:             stacklevel=2
14217:         )
14218:         self.anti_miracle_cap = anti_miracle_cap
14219:         self.logger = logging.getLogger(self.__class__.__name__)
14220:
14221:     def explain(
14222:         self,
14223:         prior: float,
14224:         signals: list[Signal],
14225:         posterior: float
14226:     ) -> PosteriorJustification:
14227:         """
14228:             Explain how each signal contributed to posterior
14229:
14230:             MANDATES:
14231:             1. Order signals by absolute marginal impact |f\224|
14232:             2. Mark discarded signals (contract violation or reconciliation failure)
14233:             3. Justify test_type in 1 line each
14234:             4. Explain anti-miracle cap application
14235:
14236:             Args:
14237:                 prior: Initial probability
14238:                 signals: List of signals with test types and likelihoods
14239:                 posterior: Final posterior probability
14240:
```

```

14241:     Returns:
14242:         PosteriorJustification with ranked signals and narrative
14243:         """
14244:         # Rank signals by marginal impact
14245:         signals_ranked = self._rank_signals_by_impact(signals)
14246:
14247:         # Identify discarded signals
14248:         discarded = [s for s in signals if not s.reconciled]
14249:
14250:         # Check if anti-miracle cap was applied
14251:         cap_applied = posterior > self.anti_miracle_cap
14252:         cap_delta = max(0, posterior - self.anti_miracle_cap) if cap_applied else ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer.__init__", "auto_param_L380_82", 0.0)
14253:
14254:         # Adjust posterior if capped
14255:         final_posterior = min(posterior, self.anti_miracle_cap)
14256:
14257:         # Generate robustness narrative
14258:         narrative = self._generate_robustness_narrative(
14259:             prior, final_posterior, signals_ranked, discarded, cap_applied, cap_delta
14260:         )
14261:
14262:         # Convert signals to dict format
14263:         ranked_dicts = [self._signal_to_dict(s) for s in signals_ranked]
14264:         discarded_dicts = [self._signal_to_dict(s) for s in discarded]
14265:
14266:         return PosteriorJustification(
14267:             prior=prior,
14268:             posterior=final_posterior,
14269:             signals_ranked=ranked_dicts,
14270:             discarded_signals=discarded_dicts,
14271:             anti_miracle_cap_applied=cap_applied,
14272:             cap_delta=cap_delta,
14273:             robustness_narrative=narrative
14274:         )
14275:
14276:     @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._rank_signals_by_impact")
14277:     def _rank_signals_by_impact(self, signals: list[Signal]) -> list[Signal]:
14278:         """Sort signals by absolute marginal impact"""
14279:         # Only rank reconciled signals
14280:         valid_signals = [s for s in signals if s.reconciled]
14281:
14282:         # Sort by |delta_posterior| descending
14283:         ranked = sorted(valid_signals, key=lambda s: abs(s.delta_posterior), reverse=True)
14284:
14285:         # Add reasons based on test type
14286:         for i, signal in enumerate(ranked):
14287:             signal.reason = self._justify_test_type(signal.test_type, i + 1)
14288:
14289:         return ranked
14290:
14291:     @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._justify_test_type")
14292:     def _justify_test_type(self, test_type: str, rank: int) -> str:
14293:         """Generate 1-line justification for test type"""
14294:         justifications = {
14295:             'Hoop': f"Rank {rank}: Necessary condition test - failure eliminates hypothesis",

```

```

14296:         'Smoking-Gun': f"Rank {rank}: Sufficient condition test - passage strongly confirms hypothesis",
14297:         'Straw-in-Wind': f"Rank {rank}: Weak evidential test - provides marginal confirmation",
14298:         'Doubly-Decisive': f"Rank {rank}: Necessary and sufficient - critical determining factor"
14299:     }
14300:     return justifications.get(test_type, f"Rank {rank}: {test_type} test applied")
14301:
14302: @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer._signal_to_dict")
14303: def _signal_to_dict(self, signal: Signal) -> dict[str, Any]:
14304:     """Convert Signal to dictionary"""
14305:     return {
14306:         'rank': 0, # Will be set by caller if needed
14307:         'test_type': signal.test_type,
14308:         'delta_posterior': signal.delta_posterior,
14309:         'kept': signal.reconciled,
14310:         'reason': signal.reason,
14311:         'likelihood': signal.likelihood,
14312:         'weight': signal.weight,
14313:         'evidence_id': signal.raw_evidence_id
14314:     }
14315:
14316: def _generate_robustness_narrative(
14317:     self,
14318:     prior: float,
14319:     posterior: float,
14320:     signals: list[Signal],
14321:     discarded: list[Signal],
14322:     cap_applied: bool,
14323:     cap_delta: float
14324: ) -> str:
14325:     """Generate 5-6 line robustness synthesis"""
14326:     narrative = f"Bayesian update from prior {prior:.3f} to posterior {posterior:.3f}. "
14327:
14328:     if signals:
14329:         top_signal = signals[0]
14330:         narrative += f"Primary driver: {top_signal.test_type} test ({top_signal.delta_posterior:.3f}). "
14331:
14332:     narrative += f"Integrated {len(signals)} reconciled signals. "
14333:
14334:     if discarded:
14335:         narrative += f"Discarded {len(discarded)} signals due to contract violations. "
14336:
14337:     if cap_applied:
14338:         narrative += f"Anti-miracle cap applied ({cap_delta:.3f} trimmed). "
14339:
14340:     # Assess robustness
14341:     if len(signals) >= 3 and not discarded:
14342:         narrative += "High robustness with diverse evidential support."
14343:     elif len(signals) >= 1:
14344:         narrative += "Moderate robustness with limited triangulation."
14345:     else:
14346:         narrative += "Low robustness - insufficient evidential base."
14347:
14348:     return narrative
14349:
14350: @calibrated_method("farfan_core.analysis.micro_prompts.BayesianPosteriorExplainer.to_json")
14351: def to_json(self, result: PosteriorJustification) -> dict[str, Any]:

```

```
14352:     """Export justification as JSON"""
14353:     return asdict(result)
14354:
14355: # =====
14356: # ANTI-MILAGRO STRESS TEST
14357: # =====
14358:
14359: @dataclass
14360: class CausalChain:
14361:     """Causal chain of steps/edges"""
14362:     steps: list[str]
14363:     edges: list[tuple[str, str]]
14364:
14365:     @calibrated_method("farfan_core.analysis.micro_prompts.CausalChain.length")
14366:     def length(self) -> int:
14367:         return len(self.steps)
14368:
14369: @dataclass
14370: class ProportionalityPattern:
14371:     """Pattern indicating proportional causal relationship"""
14372:     pattern_type: str # 'linear', 'dose-response', 'threshold', 'mechanism'
14373:     strength: float # ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.CausalChain.length", "auto_param_L501_23", 0.0)-ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.CausalChain.length", "auto_param_L501_27", 1.0)
14374:     location: str # Where in chain this appears
14375:
14376: @dataclass
14377: class StressTestResult:
14378:     """Anti-milagro stress test result"""
14379:     density: float # Patterns per chain step
14380:     simulated_drop: float # Support score drop after node removal
14381:     fragility_flag: bool # True if drop > threshold
14382:     explanation: str # 3-line explanation
14383:     pattern_coverage: float # Fraction of chain covered by patterns
14384:     missing_patterns: list[str] # Required patterns not found
14385:     timestamp: float = field(default_factory=time.time)
14386:
14387: class AntiMilagroStressTester:
14388:     """
14389:     ROLE: Structural Stress Tester [causal integrity]
14390:     GOAL: Detect dependence on non-proportional jumps
14391:
14392:     DEPRECATION WARNING: This class is a candidate for removal in 2025Q2.
14393:     Metadata:
14394:         - creation_date: 2024-01-15
14395:         - last_used_date: null
14396:         - intended_phase: future_phase_7_9
14397:         - deprecation_status: candidate_for_removal_2025Q2
14398:
14399:
14400:     def __init__(self, fragility_threshold: float = 0.3) -> None:
14401:         """
14402:             Initialize stress tester
14403:
14404:             Args:
14405:                 fragility_threshold: Support score drop threshold for fragility
14406:             """
```

```
14407:     warnings.warn(
14408:         "AntiMilagroStressTester is deprecated and scheduled for removal in 2025Q2. "
14409:         "No usage detected as of February 2025. Intended for future_phase_7_9. "
14410:         "Metadata: creation_date=2024-01-15, last_used_date=null, "
14411:         "intended_phase=future_phase_7_9, deprecation_status=candidate_for_removal_2025Q2",
14412:         DeprecationWarning,
14413:         stacklevel=2
14414:     )
14415:     self.fragility_threshold = fragility_threshold
14416:     self.logger = logging.getLogger(self.__class__.__name__)
14417:
14418:     def stress_test(
14419:         self,
14420:         causal_chain: CausalChain,
14421:         proportionality_patterns: list[ProportionalityPattern],
14422:         missing_patterns: list[str]
14423:     ) -> StressTestResult:
14424:         """
14425:             Stress test causal chain for structural fragility
14426:
14427:             MANDATES:
14428:             1. Evaluate pattern density vs chain length
14429:             2. Simulate weak node removal and recalculate support
14430:             3. Flag fragility if drop > 1\204
14431:
14432:             Args:
14433:                 causal_chain: Chain of causal steps
14434:                 proportionality_patterns: Detected proportionality patterns
14435:                 missing_patterns: Required patterns not found
14436:
14437:             Returns:
14438:                 StressTestResult with fragility assessment
14439:             """
14440:             # 1. Calculate pattern density
14441:             density = self._calculate_pattern_density(causal_chain, proportionality_patterns)
14442:
14443:             # 2. Simulate node removal
14444:             simulated_drop = self._simulate_node_removal(causal_chain, proportionality_patterns)
14445:
14446:             # 3. Check fragility
14447:             fragility_flag = simulated_drop > self.fragility_threshold
14448:
14449:             # Calculate pattern coverage
14450:             coverage = self._calculate_pattern_coverage(causal_chain, proportionality_patterns)
14451:
14452:             # Generate explanation
14453:             explanation = self._generate_explanation(density, simulated_drop, fragility_flag)
14454:
14455:             return StressTestResult(
14456:                 density=density,
14457:                 simulated_drop=simulated_drop,
14458:                 fragility_flag=fragility_flag,
14459:                 explanation=explanation,
14460:                 pattern_coverage=coverage,
14461:                 missing_patterns=missing_patterns
14462:             )
```

```

14463:
14464:     def _calculate_pattern_density(
14465:         self, chain: CausalChain, patterns: list[ProportionalityPattern]
14466:     ) -> float:
14467:         """Calculate patterns per chain step"""
14468:         if chain.length() == 0:
14469:             return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L582_19", 0.0)
14470:         return len(patterns) / chain.length()
14471:
14472:     def _calculate_pattern_coverage(
14473:         self, chain: CausalChain, patterns: list[ProportionalityPattern]
14474:     ) -> float:
14475:         """Calculate fraction of chain covered by patterns"""
14476:         if chain.length() == 0:
14477:             return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L590_19", 0.0)
14478:
14479:         # Count unique steps covered by patterns
14480:         covered_steps = set()
14481:         for pattern in patterns:
14482:             # Extract step indices from pattern location
14483:             # This is simplified - actual implementation would parse location
14484:             covered_steps.add(pattern.location)
14485:
14486:         return len(covered_steps) / chain.length()
14487:
14488:     def _simulate_node_removal(
14489:         self, chain: CausalChain, patterns: list[ProportionalityPattern]
14490:     ) -> float:
14491:         """Simulate removal of weak nodes and measure support drop"""
14492:         if not patterns or chain.length() == 0:
14493:             return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L606_19", 1.0) # Maximum drop if no patterns
14494:
14495:         # Calculate baseline support score
14496:         baseline_support = self._calculate_support_score(patterns)
14497:
14498:         # Identify weak patterns (bottom 25% by strength)
14499:         if len(patterns) > 1:
14500:             strengths = [p.strength for p in patterns]
14501:             threshold = np.percentile(strengths, 25)
14502:             strong_patterns = [p for p in patterns if p.strength > threshold]
14503:         else:
14504:             strong_patterns = patterns
14505:
14506:         # Calculate support without weak patterns
14507:         reduced_support = self._calculate_support_score(strong_patterns)
14508:
14509:         # Calculate drop
14510:         if baseline_support == 0:
14511:             return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L624_19", 0.0)
14512:
14513:         drop = (baseline_support - reduced_support) / baseline_support
14514:         return max(ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L627_19", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.__init__", "auto_param_L627_28", 1.0), drop)) # Clamp to [0, 1]
14515:
14516:     @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._calculate_support_score")

```

```

14517:     def _calculate_support_score(self, patterns: list[ProportionalityPattern]) -> float:
14518:         """Calculate overall support score from patterns"""
14519:         if not patterns:
14520:             return ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._calculate_support_score", "auto_param_L633_19", 0.0)
14521:
14522:         # Weighted average of pattern strengths
14523:         total_weight = sum(p.strength for p in patterns)
14524:         return total_weight / len(patterns)
14525:
14526:     @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester._generate_explanation")
14527:     def _generate_explanation(self, density: float, drop: float, fragility: bool) -> str:
14528:         """Generate 3-line explanation"""
14529:         explanation = f"Pattern density: {density:.2f} patterns/step. "
14530:         explanation += f"Simulated node removal causes {drop:.1%} support drop. "
14531:
14532:         if fragility:
14533:             explanation += "FRAGILITY DETECTED: Drop exceeds threshold, indicating structural weakness."
14534:         else:
14535:             explanation += "Robust structure: Support maintained under stress."
14536:
14537:     return explanation
14538:
14539:     @calibrated_method("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json")
14540:     def to_json(self, result: StressTestResult) -> dict[str, Any]:
14541:         """Export stress test result as JSON"""
14542:         return asdict(result)
14543:
14544: # =====
14545: # CONVENIENCE FUNCTIONS
14546: # =====
14547:
14548:     def create_provenance_auditor(
14549:         p95_latency: float | None = None,
14550:         contracts: dict[str, dict[str, Any]] | None = None
14551:     ) -> ProvenanceAuditor:
14552:         """Factory function for ProvenanceAuditor"""
14553:         return ProvenanceAuditor(p95_latency, contracts)
14554:
14555:     def create_posterior_explainer(anti_miracle_cap: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json", "auto_param_L668_57", 0.95)) -> BayesianPosteriorExplainer:
14556:         """Factory function for BayesianPosteriorExplainer"""
14557:         return BayesianPosteriorExplainer(anti_miracle_cap)
14558:
14559:     def create_stress_tester(fragility_threshold: float = ParameterLoaderV2.get("farfan_core.analysis.micro_prompts.AntiMilagroStressTester.to_json", "auto_param_L672_54", 0.3)) -> AntiMilagroStressTester:
14560:         """Factory function for AntiMilagroStressTester"""
14561:         return AntiMilagroStressTester(fragility_threshold)
14562:
14563:
14564:
14565: =====
14566: FILE: src/farfan_pipeline/analysis/recommendation_engine.py
14567: =====
14568:
14569: # recommendation_engine.py - Rule-Based Recommendation Engine
14570: """

```

```
14571: Recommendation Engine - Multi-Level Rule-Based Recommendations
14572: =====
14573:
14574: This module implements a rule-based recommendation engine that:
14575: 1. Loads and validates recommendation rules from JSON files
14576: 2. Evaluates conditions against score data at MICRO, MESO, and MACRO levels
14577: 3. Generates actionable recommendations with specific interventions
14578: 4. Renders templates with context-specific variable substitution
14579:
14580: Supports three levels of recommendations:
14581: - MICRO: Question-level recommendations (PA-DIM combinations)
14582: - MESO: Cluster-level recommendations (CL01-CL04)
14583: - MACRO: Plan-level strategic recommendations
14584:
14585: Author: Integration Team
14586: Version: 2.0.0
14587: Python: 3.10+
14588: """
14589:
14590: import logging
14591: import re
14592: from dataclasses import asdict, dataclass, field
14593: from datetime import datetime, timezone
14594: from pathlib import Path
14595: from typing import Any
14596:
14597: import jsonschema
14598: from farfan_pipeline.core.parameters import ParameterLoaderV2
14599: from farfan_pipeline.core.calibration.decorators import calibrated_method
14600:
14601: logger = logging.getLogger(__name__)
14602:
14603: _REQUIRED_ENHANCED_FEATURES = {
14604:     "template_parameterization",
14605:     "execution_logic",
14606:     "measurable_indicators",
14607:     "unambiguous_time_horizons",
14608:     "testable_verification",
14609:     "cost_tracking",
14610:     "authority_mapping",
14611: }
14612:
14613: # =====
14614: # DATA STRUCTURES FOR RECOMMENDATIONS
14615: # =====
14616:
14617: @dataclass
14618: class Recommendation:
14619: """
14620:     Structured recommendation with full intervention details.
14621:
14622:     Supports both v1.0 (simple) and v2.0 (enhanced with 7 advanced features):
14623:         1. Template parameterization
14624:         2. Execution logic
14625:         3. Measurable indicators
14626:         4. Unambiguous time horizons
```

```
14627:     5. Testable verification
14628:     6. Cost tracking
14629:     7. Authority mapping
14630: """
14631:     rule_id: str
14632:     level: str # MICRO, MESO, or MACRO
14633:     problem: str
14634:     intervention: str
14635:     indicator: dict[str, Any]
14636:     responsible: dict[str, Any]
14637:     horizon: dict[str, Any] # Changed from Dict[str, str] to support enhanced fields
14638:     verification: list[Any] # Changed from List[str] to support structured verification
14639:     metadata: dict[str, Any] = field(default_factory=dict)
14640:
14641:     # Enhanced fields (v2.0) - optional for backward compatibility
14642:     execution: dict[str, Any] | None = None
14643:     budget: dict[str, Any] | None = None
14644:     template_id: str | None = None
14645:     template_params: dict[str, Any] | None = None
14646:
14647:     @calibrated_method("farfan_core.analysis.recommendation_engine.Recommendation.to_dict")
14648:     def to_dict(self) -> dict[str, Any]:
14649:         """Convert to dictionary for JSON serialization"""
14650:         result = asdict(self)
14651:         # Remove None values for cleaner output
14652:         return {k: v for k, v in result.items() if v is not None}
14653:
14654: @dataclass
14655: class RecommendationSet:
14656: """
14657: Collection of recommendations with metadata
14658: """
14659:     level: str
14660:     recommendations: list[Recommendation]
14661:     generated_at: str
14662:     total_rules_evaluated: int
14663:     rules_matched: int
14664:     metadata: dict[str, Any] = field(default_factory=dict)
14665:
14666:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationSet.to_dict")
14667:     def to_dict(self) -> dict[str, Any]:
14668:         """Convert to dictionary for JSON serialization"""
14669:         return {
14670:             'level': self.level,
14671:             'recommendations': [r.to_dict() for r in self.recommendations],
14672:             'generated_at': self.generated_at,
14673:             'total_rules_evaluated': self.total_rules_evaluated,
14674:             'rules_matched': self.rules_matched,
14675:             'metadata': self.metadata
14676:         }
14677:
14678: # =====
14679: # RECOMMENDATION ENGINE
14680: # =====
14681:
14682: class RecommendationEngine:
```

```
14683: """
14684:     Core recommendation engine that evaluates rules and generates recommendations.
14685:
14686:     Uses canonical notation for dimension and policy area validation.
14687:
14688:
14689:     def __init__(
14690:         self,
14691:             rules_path: str = "config/recommendation_rules_enhanced.json",
14692:             schema_path: str = "rules/recommendation_rules.schema.json",
14693:             questionnaire_provider=None,
14694:             orchestrator=None
14695:         ) -> None:
14696:             """
14697:                 Initialize recommendation engine
14698:
14699:             Args:
14700:                 rules_path: Path to recommendation rules JSON file
14701:                 schema_path: Path to JSON schema for validation
14702:                 questionnaire_provider: QuestionnaireResourceProvider instance (injected via DI)
14703:                 orchestrator: Orchestrator instance for accessing thresholds and patterns
14704:
14705:             ARCHITECTURAL NOTE: Thresholds should come from questionnaire monolith
14706:             via QuestionnaireResourceProvider, not from hardcoded values.
14707:             """
14708:             self.rules_path = Path(rules_path)
14709:             self.schema_path = Path(schema_path)
14710:             self.questionnaire_provider = questionnaire_provider
14711:             self.orchestrator = orchestrator
14712:             self.rules: dict[str, Any] = {}
14713:             self.schema: dict[str, Any] = {}
14714:             self.rules_by_level: dict[str, list[dict[str, Any]]] = {
14715:                 'MICRO': [],
14716:                 'MESO': [],
14717:                 'MACRO': []
14718:             }
14719:
14720:             # Load canonical notation for validation
14721:             self._load_canonical_notation()
14722:
14723:             # Load rules and schema
14724:             self._load_schema()
14725:             self._load_rules()
14726:
14727:             logger.info(
14728:                 f"Recommendation engine initialized with "
14729:                 f"{len(self.rules_by_level['MICRO'])} MICRO, "
14730:                 f"{len(self.rules_by_level['MESO'])} MESO, "
14731:                 f"{len(self.rules_by_level['MACRO'])} MACRO rules"
14732:             )
14733:
14734:             @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_canonical_notation")
14735:             def _load_canonical_notation(self) -> None:
14736:                 """Load canonical notation for validation"""
14737:                 try:
14738:                     from farfan_pipeline.core.canonical_notation import get_all_dimensions, get_all_policy_areas
```

```
14739:         self.canonical_dimensions = get_all_dimensions()
14740:         self.canonical_policy_areas = get_all_policy_areas()
14741:         logger.info(
14742:             f"Canonical notation loaded: {len(self.canonical_dimensions)} dimensions, "
14743:             f"{len(self.canonical_policy_areas)} policy areas"
14744:         )
14745:     except Exception as e:
14746:         logger.warning(f"Could not load canonical notation: {e}")
14747:         self.canonical_dimensions = {}
14748:         self.canonical_policy_areas = {}
14749:
14750:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_schema")
14751:     def _load_schema(self) -> None:
14752:         """Load JSON schema for rule validation"""
14753:         # Delegate to factory for I/O operation
14754:         from farfan_pipeline.analysis.factory import load_json
14755:
14756:         try:
14757:             self.schema = load_json(self.schema_path)
14758:             logger.info(f"Loaded recommendation rules schema from {self.schema_path}")
14759:         except Exception as e:
14760:             logger.error(f"Failed to load schema: {e}")
14761:             raise
14762:
14763:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._load_rules")
14764:     def _load_rules(self) -> None:
14765:         """Load and validate recommendation rules"""
14766:         # Delegate to factory for I/O operation
14767:         from farfan_pipeline.analysis.factory import load_json
14768:
14769:         try:
14770:             self.rules = load_json(self.rules_path)
14771:
14772:             # Validate against schema
14773:             jsonschema.validate(instance=self.rules, schema=self.schema)
14774:             self._validate_ruleset_metadata()
14775:
14776:             # Organize rules by level
14777:             for rule in self.rules.get('rules', []):
14778:                 self._validate_rule(rule)
14779:                 level = rule.get('level')
14780:                 if level in self.rules_by_level:
14781:                     self.rules_by_level[level].append(rule)
14782:
14783:             logger.info(f"Loaded and validated {len(self.rules.get('rules', []))} rules from {self.rules_path}")
14784:         except jsonschema.ValidationError as e:
14785:             logger.error(f"Rule validation failed: {e.message}")
14786:             raise
14787:         except Exception as e:
14788:             logger.error(f"Failed to load rules: {e}")
14789:             raise
14790:
14791:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine.reload_rules")
14792:     def reload_rules(self) -> None:
14793:         """Reload rules from disk (useful for hot-reloading)"""
14794:         self.rules_by_level = {'MICRO': [], 'MESO': [], 'MACRO': []}
```

```
14795:         self._load_rules()
14796:
14797:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith")
14798:     def get_thresholds_from_monolith(self) -> dict[str, Any]:
14799:         """
14800:             Get scoring thresholds from questionnaire monolith.
14801:
14802:             Returns:
14803:                 Dictionary of thresholds by question_id or default thresholds
14804:
14805:             ARCHITECTURAL NOTE: This method demonstrates proper access to
14806:             questionnaire data via QuestionnaireResourceProvider, not direct I/O.
14807:             """
14808:             if self.questionnaire_provider is None:
14809:                 logger.warning("No questionnaire provider attached, using default thresholds")
14810:                 return {
14811:                     'default_micro_threshold': 2.0,
14812:                     'default_meso_threshold': 55.0,
14813:                     'default_macro_threshold': 65.0
14814:                 }
14815:
14816:             # Get questionnaire data via provider
14817:             questionnaire_data = self.questionnaire_provider.get_data()
14818:
14819:             # Extract thresholds from monolith structure
14820:             thresholds = {}
14821:             blocks = questionnaire_data.get('blocks', {})
14822:             micro_questions = blocks.get('micro_questions', [])
14823:
14824:             for question in micro_questions:
14825:                 question_id = question.get('question_id')
14826:                 scoring_info = question.get('scoring', {})
14827:                 threshold = scoring_info.get('threshold')
14828:
14829:                 if question_id and threshold is not None:
14830:                     thresholds[question_id] = threshold
14831:
14832:             logger.info(f"Loaded {len(thresholds)} thresholds from questionnaire monolith")
14833:             return thresholds
14834:
14835:             # =====
14836:             # MICRO LEVEL RECOMMENDATIONS
14837:             # =====
14838:
14839:             def generate_micro_recommendations(
14840:                 self,
14841:                 scores: dict[str, float],
14842:                 context: dict[str, Any] | None = None
14843:             ) -> RecommendationSet:
14844:                 """
14845:                     Generate MICRO-level recommendations based on PA-DIM scores
14846:
14847:                     Args:
14848:                         scores: Dictionary mapping "PA##-DIM##" to scores (ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_th
resholds_from_monolith", "auto_param_L279_63", 0.0)-3.0)
14849:                         context: Additional context for template rendering
```

```
14850:  
14851:     Returns:  
14852:         RecommendationSet with matched recommendations  
14853:     """  
14854:     recommendations = []  
14855:     rules_evaluated = 0  
14856:  
14857:     for rule in self.rules_by_level['MICRO']:  
14858:         rules_evaluated += 1  
14859:  
14860:         # Extract condition  
14861:         when = rule.get('when', {})  
14862:         pa_id = when.get('pa_id')  
14863:         dim_id = when.get('dim_id')  
14864:         score_lt = when.get('score_lt')  
14865:  
14866:         # Build score key  
14867:         score_key = f"{pa_id}-{dim_id}"  
14868:  
14869:         # Check if condition matches  
14870:         if score_key in scores and scores[score_key] < score_lt:  
14871:             # Render template  
14872:             template = rule.get('template', {})  
14873:             rendered = self._render_micro_template(template, pa_id, dim_id, context)  
14874:  
14875:             # Create recommendation with enhanced fields (v2.0) if available  
14876:             rec = Recommendation(  
14877:                 rule_id=rule.get('rule_id'),  
14878:                 level='MICRO',  
14879:                 problem=rendered['problem'],  
14880:                 intervention=rendered['intervention'],  
14881:                 indicator=rendered['indicator'],  
14882:                 responsible=rendered['responsible'],  
14883:                 horizon=rendered['horizon'],  
14884:                 verification=rendered['verification'],  
14885:                 metadata={  
14886:                     'score_key': score_key,  
14887:                     'actual_score': scores[score_key],  
14888:                     'threshold': score_lt,  
14889:                     'gap': score_lt - scores[score_key]  
14890:                 },  
14891:                 # Enhanced fields (v2.0)  
14892:                 execution=rule.get('execution'),  
14893:                 budget=rule.get('budget'),  
14894:                 template_id=rendered.get('template_id'),  
14895:                 template_params=rendered.get('template_params')  
14896:             )  
14897:             recommendations.append(rec)  
14898:  
14899:     return RecommendationSet(  
14900:         level='MICRO',  
14901:         recommendations=recommendations,  
14902:         generated_at=datetime.now(timezone.utc).isoformat(),  
14903:         total_rules_evaluated=rules_evaluated,  
14904:         rules_matched=len(recommendations)  
14905:     )
```

```
14906:  
14907:     def _render_micro_template(  
14908:         self,  
14909:         template: dict[str, Any],  
14910:         pa_id: str,  
14911:         dim_id: str,  
14912:         context: dict[str, Any] | None = None  
14913:     ) -> dict[str, Any]:  
14914:         """  
14915:             Render MICRO template with variable substitution  
14916:  
14917:             Variables supported:  
14918:             - {{PAxx}}: Policy area (e.g., PA01)  
14919:             - {{DIMxx}}: Dimension (e.g., DIM01)  
14920:             - {{Q##}}: Question number (from context)  
14921:         """  
14922:         ctx = context or {}  
14923:  
14924:         substitutions = {  
14925:             'PAxx': pa_id,  
14926:             'DIMxx': dim_id,  
14927:             'pa_id': pa_id,  
14928:             'dim_id': dim_id,  
14929:         }  
14930:  
14931:         question_hint = ctx.get('question_id')  
14932:         template_params = template.get("template_params", {}) if isinstance(template, dict) else {}  
14933:         if isinstance(template_params, dict):  
14934:             for key, value in template_params.items():  
14935:                 if isinstance(value, str):  
14936:                     substitutions.setdefault(key, value)  
14937:                     substitutions.setdefault(key.upper(), value)  
14938:                 if key == 'question_id':  
14939:                     question_hint = value  
14940:  
14941:             if isinstance(question_hint, str):  
14942:                 substitutions.setdefault(question_hint, question_hint)  
14943:                 substitutions.setdefault('question_id', question_hint)  
14944:                 substitutions.setdefault('Q001', question_hint)  
14945:  
14946:             for key, value in ctx.items():  
14947:                 if isinstance(value, str):  
14948:                     substitutions.setdefault(key, value)  
14949:  
14950:         return self._render_template(template, substitutions)  
14951:  
14952: # ======  
14953: # MESO LEVEL RECOMMENDATIONS  
14954: # ======  
14955:  
14956:     def generate_meso_recommendations(  
14957:         self,  
14958:         cluster_data: dict[str, Any],  
14959:         context: dict[str, Any] | None = None  
14960:     ) -> RecommendationSet:  
14961:         """
```

```
14962:     Generate MESO-level recommendations based on cluster performance
14963:
14964:     Args:
14965:         cluster_data: Dictionary with cluster metrics:
14966:             {
14967:                 'CL01': {'score': 75.0, 'variance': ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_threshold
s_from_monolith", "auto_param_L398_56", 0.15), 'weak_pa': 'PA02'},
14968:                 'CL02': {'score': 62.0, 'variance': ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_threshold
s_from_monolith", "auto_param_L399_56", 0.22), 'weak_pa': 'PA05'},
14969:             ...
14970:         }
14971:         context: Additional context for template rendering
14972:
14973:     Returns:
14974:         RecommendationSet with matched recommendations
14975:     """
14976:     recommendations = []
14977:     rules_evaluated = 0
14978:
14979:     for rule in self.rules_by_level['MESO']:
14980:         rules_evaluated += 1
14981:
14982:         # Extract condition
14983:         when = rule.get('when', {})
14984:         cluster_id = when.get('cluster_id')
14985:         score_band = when.get('score_band')
14986:         variance_level = when.get('variance_level')
14987:         variance_threshold = when.get('variance_threshold')
14988:         weak_pa_id = when.get('weak_pa_id')
14989:
14990:         # Get cluster data
14991:         cluster = cluster_data.get(cluster_id, {})
14992:         cluster_score = cluster.get('score', 0)
14993:         cluster_variance = cluster.get('variance', 0)
14994:         cluster_weak_pa = cluster.get('weak_pa')
14995:
14996:         # Check conditions
14997:         if not self._check_meso_conditions(
14998:             cluster_score, cluster_variance, cluster_weak_pa,
14999:             score_band, variance_level, variance_threshold, weak_pa_id
15000:         ):
15001:             continue
15002:
15003:         # Render template
15004:         template = rule.get('template', {})
15005:         rendered = self._render_meso_template(template, cluster_id, context)
15006:
15007:         # Create recommendation with enhanced fields (v2.0) if available
15008:         rec = Recommendation(
15009:             rule_id=rule.get('rule_id'),
15010:             level='MESO',
15011:             problem=rendered['problem'],
15012:             intervention=rendered['intervention'],
15013:             indicator=rendered['indicator'],
15014:             responsible=rendered['responsible'],
15015:             horizon=rendered['horizon'],
```

```

15016:             verification=rendered['verification'],
15017:             metadata={
15018:                 'cluster_id': cluster_id,
15019:                 'score': cluster_score,
15020:                 'score_band': score_band,
15021:                 'variance': cluster_variance,
15022:                 'variance_level': variance_level,
15023:                 'weak_pa': cluster_weak_pa
15024:             },
15025:             # Enhanced fields (v2.0)
15026:             execution=rule.get('execution'),
15027:             budget=rule.get('budget'),
15028:             template_id=rendered.get('template_id'),
15029:             template_params=rendered.get('template_params')
15030:         )
15031:         recommendations.append(rec)
15032:
15033:     return RecommendationSet(
15034:         level='MESO',
15035:         recommendations=recommendations,
15036:         generated_at=datetime.now(timezone.utc).isoformat(),
15037:         total_rules_evaluated=rules_evaluated,
15038:         rules_matched=len(recommendations)
15039:     )
15040:
15041:     def _check_meso_conditions(
15042:         self,
15043:         score: float,
15044:         variance: float,
15045:         weak_pa: str | None,
15046:         score_band: str,
15047:         variance_level: str,
15048:         variance_threshold: float | None,
15049:         weak_pa_id: str | None
15050:     ) -> bool:
15051:         """Check if MESO conditions are met"""
15052:         # Check score band
15053:         if score_band == 'BAJO' and score >= 55 or score_band == 'MEDIO' and (score < 55 or score >= 75) or score_band == 'ALTO' and score < 75:
15054:             return False
15055:
15056:         # Check variance level
15057:         if variance_level == 'BAJA' and variance >= ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_52", 0.08) or variance_level == 'MEDIA' and (variance < ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_102", 0.08) or variance >= ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L488_122", 0.18)):
15058:             return False
15059:         elif variance_level == 'ALTA':
15060:             if variance_threshold and variance < variance_threshold / 100 or not variance_threshold and variance < ParameterLoaderV2.get("farfan_core.analysis.recommendation_engine.RecommendationEngine.get_thresholds_from_monolith", "auto_param_L491_115", 0.18):
15061:                 return False
15062:
15063:         # Check weak PA if specified
15064:         return not (weak_pa_id and weak_pa != weak_pa_id)
15065:
15066:     def _render_meso_template(
15067:         self,

```

```

15068:         template: dict[str, Any],
15069:         cluster_id: str,
15070:         context: dict[str, Any] | None = None
15071:     ) -> dict[str, Any]:
15072:         """Render MESO template with variable substitution"""
15073:
15074:         substitutions = {
15075:             'cluster_id': cluster_id,
15076:         }
15077:
15078:         if isinstance(template, dict):
15079:             params = template.get('template_params', {})
15080:             if isinstance(params, dict):
15081:                 for key, value in params.items():
15082:                     if isinstance(value, str):
15083:                         substitutions.setdefault(key, value)
15084:                         substitutions.setdefault(key.upper(), value)
15085:
15086:             if context:
15087:                 for key, value in context.items():
15088:                     if isinstance(value, str):
15089:                         substitutions.setdefault(key, value)
15090:
15091:         return self._render_template(template, substitutions)
15092:
15093: # =====
15094: # MACRO LEVEL RECOMMENDATIONS
15095: # =====
15096:
15097: def generate_macro_recommendations(
15098:     self,
15099:     macro_data: dict[str, Any],
15100:     context: dict[str, Any] | None = None
15101: ) -> RecommendationSet:
15102:     """
15103:     Generate MACRO-level strategic recommendations
15104:
15105:     Args:
15106:         macro_data: Dictionary with plan-level metrics:
15107:             {
15108:                 'macro_band': 'SATISFACTORIO',
15109:                 'clusters_below_target': ['CL02', 'CL03'],
15110:                 'variance_alert': 'MODERADA',
15111:                 'priority_micro_gaps': ['PA01-DIM05', 'PA04-DIM04']
15112:             }
15113:         context: Additional context for template rendering
15114:
15115:     Returns:
15116:         RecommendationSet with matched recommendations
15117:     """
15118:     recommendations = []
15119:     rules_evaluated = 0
15120:
15121:     for rule in self.rules_by_level['MACRO']:
15122:         rules_evaluated += 1
15123:

```

```

15124:     # Extract condition
15125:     when = rule.get('when', {})
15126:     macro_band = when.get('macro_band')
15127:     clusters_below = set(when.get('clusters_below_target', []))
15128:     variance_alert = when.get('variance_alert')
15129:     priority_gaps = set(when.get('priority_micro_gaps', []))
15130:
15131:     # Get macro data
15132:     actual_band = macro_data.get('macro_band')
15133:     actual_clusters = set(macro_data.get('clusters_below_target', []))
15134:     actual_variance = macro_data.get('variance_alert')
15135:     actual_gaps = set(macro_data.get('priority_micro_gaps', []))
15136:
15137:     # Check conditions
15138:     if macro_band and macro_band != actual_band:
15139:         continue
15140:     if variance_alert and variance_alert != actual_variance:
15141:         continue
15142:
15143:     # Check if clusters match (subset or exact match)
15144:     if clusters_below and not clusters_below.issubset(actual_clusters):
15145:         # For MACRO, we want exact match or the rule's clusters to be present
15146:         if clusters_below != actual_clusters and not actual_clusters.issubset(clusters_below):
15147:             continue
15148:
15149:     # Check if priority gaps match (subset)
15150:     if priority_gaps and not priority_gaps.issubset(actual_gaps):
15151:         continue
15152:
15153:     # Render template
15154:     template = rule.get('template', {})
15155:     rendered = self._render_macro_template(template, context)
15156:
15157:     # Create recommendation with enhanced fields (v2.0) if available
15158:     rec = Recommendation(
15159:         rule_id=rule.get('rule_id'),
15160:         level='MACRO',
15161:         problem=rendered['problem'],
15162:         intervention=rendered['intervention'],
15163:         indicator=rendered['indicator'],
15164:         responsible=rendered['responsible'],
15165:         horizon=rendered['horizon'],
15166:         verification=rendered['verification'],
15167:         metadata={
15168:             'macro_band': actual_band,
15169:             'clusters_below_target': list(actual_clusters),
15170:             'variance_alert': actual_variance,
15171:             'priority_micro_gaps': list(actual_gaps)
15172:         },
15173:         # Enhanced fields (v2.0)
15174:         execution=rule.get('execution'),
15175:         budget=rule.get('budget'),
15176:         template_id=rendered.get('template_id'),
15177:         template_params=rendered.get('template_params')
15178:     )
15179:     recommendations.append(rec)

```

```
15180:
15181:     return RecommendationSet(
15182:         level='MACRO',
15183:         recommendations=recommendations,
15184:         generated_at=datetime.now(timezone.utc).isoformat(),
15185:         total_rules_evaluated=rules_evaluated,
15186:         rules_matched=len(recommendations)
15187:     )
15188:
15189:     def _render_macro_template(
15190:         self,
15191:         template: dict[str, Any],
15192:         context: dict[str, Any] | None = None
15193:     ) -> dict[str, Any]:
15194:         """Render MACRO template with variable substitution"""
15195:
15196:         substitutions = {}
15197:
15198:         if context:
15199:             for key, value in context.items():
15200:                 if isinstance(value, str):
15201:                     substitutions.setdefault(key, value)
15202:
15203:         if isinstance(template, dict):
15204:             params = template.get('template_params', {})
15205:             if isinstance(params, dict):
15206:                 for key, value in params.items():
15207:                     if isinstance(value, str):
15208:                         substitutions.setdefault(key, value)
15209:                         substitutions.setdefault(key.upper(), value)
15210:
15211:             return self._render_template(template, substitutions)
15212:
15213: # =====
15214: # UTILITY METHODS
15215: # =====
15216:
15217: @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._substitute_variables")
15218: def _substitute_variables(self, text: str, substitutions: dict[str, str]) -> str:
15219:     """
15220:         Substitute variables in text using {{variable}} syntax
15221:
15222:         Args:
15223:             text: Text with variables
15224:             substitutions: Dictionary of variable_name -> value
15225:
15226:         Returns:
15227:             Text with variables substituted
15228:     """
15229:     result = text
15230:     for var, value in substitutions.items():
15231:         pattern = r'\{\{ ' + re.escape(var) + r'\}\}\''
15232:         result = re.sub(pattern, value, result)
15233:     return result
15234:
15235: @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._render_template")
```

```
15236:     def _render_template(self, template: dict[str, Any], substitutions: dict[str, str]) -> dict[str, Any]:
15237:         """Recursively render a template applying substitutions to nested structures."""
15238:
15239:         def render_value(value: Any) -> Any:
15240:             if isinstance(value, str):
15241:                 return self._substitute_variables(value, substitutions)
15242:             if isinstance(value, list):
15243:                 return [render_value(item) for item in value]
15244:             if isinstance(value, dict):
15245:                 return {k: render_value(v) for k, v in value.items()}
15246:             return value
15247:
15248:         return render_value(template)
15249:
15250: # =====
15251: # VALIDATION UTILITIES
15252: # =====
15253:
15254: @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_rule")
15255: def _validate_rule(self, rule: dict[str, Any]) -> None:
15256:     """Apply structural validation to guarantee rigorous recommendations."""
15257:     rule_id = rule.get('rule_id')
15258:     if not isinstance(rule_id, str) or not rule_id.strip():
15259:         raise ValueError("Recommendation rule missing rule_id")
15260:
15261:     level = rule.get('level')
15262:     if level not in self.rules_by_level:
15263:         raise ValueError(f"Rule {rule_id} declares unsupported level: {level}")
15264:
15265:     when = rule.get('when', {})
15266:     if not isinstance(when, dict):
15267:         raise ValueError(f"Rule {rule_id} has invalid 'when' definition")
15268:
15269:     if level == 'MICRO':
15270:         self._validate_micro_when(rule_id, when)
15271:     elif level == 'MESO':
15272:         self._validate_meso_when(rule_id, when)
15273:     elif level == 'MACRO':
15274:         self._validate_macro_when(rule_id, when)
15275:
15276:     template = rule.get('template')
15277:     if not isinstance(template, dict):
15278:         raise ValueError(f"Rule {rule_id} lacks a structured template")
15279:
15280:     self._validate_template(rule_id, template, level)
15281:
15282:     execution = rule.get('execution')
15283:     if execution is None:
15284:         raise ValueError(f"Rule {rule_id} is missing execution block required for enhanced rules")
15285:     self._validate_execution(rule_id, execution)
15286:
15287:     budget = rule.get('budget')
15288:     if budget is None:
15289:         raise ValueError(f"Rule {rule_id} is missing budget block required for enhanced rules")
15290:     self._validate_budget(rule_id, budget)
15291:
```

```
15292:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_micro_when")
15293:     def _validate_micro_when(self, rule_id: str, when: dict[str, Any]) -> None:
15294:         required_keys = ('pa_id', 'dim_id', 'score_lt')
15295:         for key in required_keys:
15296:             if key not in when:
15297:                 raise ValueError(f"Rule {rule_id} missing '{key}' in MICRO condition")
15298:
15299:         pa_id = when['pa_id']
15300:         dim_id = when['dim_id']
15301:         if not isinstance(pa_id, str) or not pa_id.strip():
15302:             raise ValueError(f"Rule {rule_id} has invalid pa_id")
15303:         if not isinstance(dim_id, str) or not dim_id.strip():
15304:             raise ValueError(f"Rule {rule_id} has invalid dim_id")
15305:
15306:         score_lt = when['score_lt']
15307:         if not self._is_number(score_lt):
15308:             raise ValueError(f"Rule {rule_id} has non-numeric MICRO threshold")
15309:         if not 0 <= float(score_lt) <= 3:
15310:             raise ValueError(f"Rule {rule_id} MICRO threshold must be between 0 and 3")
15311:
15312:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_meso_when")
15313:     def _validate_meso_when(self, rule_id: str, when: dict[str, Any]) -> None:
15314:         cluster_id = when.get('cluster_id')
15315:         if not isinstance(cluster_id, str) or not cluster_id.strip():
15316:             raise ValueError(f"Rule {rule_id} missing cluster_id for MESO condition")
15317:
15318:         condition_counter = 0
15319:
15320:         score_band = when.get('score_band')
15321:         if score_band is not None:
15322:             if score_band not in {'BAJO', 'MEDIO', 'ALTO'}:
15323:                 raise ValueError(f"Rule {rule_id} has invalid MESO score_band")
15324:             condition_counter += 1
15325:
15326:         variance_level = when.get('variance_level')
15327:         if variance_level is not None:
15328:             if variance_level not in {'BAJA', 'MEDIA', 'ALTA'}:
15329:                 raise ValueError(f"Rule {rule_id} has invalid MESO variance_level")
15330:             condition_counter += 1
15331:
15332:         variance_threshold = when.get('variance_threshold')
15333:         if variance_threshold is not None and not self._is_number(variance_threshold):
15334:             raise ValueError(f"Rule {rule_id} has non-numeric variance_threshold")
15335:
15336:         weak_pa_id = when.get('weak_pa_id')
15337:         if weak_pa_id is not None:
15338:             if not isinstance(weak_pa_id, str) or not weak_pa_id.strip():
15339:                 raise ValueError(f"Rule {rule_id} has invalid weak_pa_id")
15340:             condition_counter += 1
15341:
15342:         if condition_counter == 0:
15343:             raise ValueError(
15344:                 f"Rule {rule_id} must specify at least one discriminant condition for MESO"
15345:             )
15346:
15347:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_macro_when")
```

```

15348:     def _validate_macro_when(self, rule_id: str, when: dict[str, Any]) -> None:
15349:         discriminants = 0
15350:
15351:         macro_band = when.get('macro_band')
15352:         if macro_band is not None:
15353:             if not isinstance(macro_band, str) or not macro_band.strip():
15354:                 raise ValueError(f"Rule {rule_id} has invalid macro_band")
15355:             discriminants += 1
15356:
15357:         clusters = when.get('clusters_below_target')
15358:         if clusters is not None:
15359:             if not isinstance(clusters, list) or not clusters:
15360:                 raise ValueError(f"Rule {rule_id} must declare non-empty clusters_below_target")
15361:             if not all(isinstance(item, str) and item.strip() for item in clusters):
15362:                 raise ValueError(f"Rule {rule_id} has invalid cluster identifiers")
15363:             discriminants += 1
15364:
15365:         variance_alert = when.get('variance_alert')
15366:         if variance_alert is not None:
15367:             if not isinstance(variance_alert, str) or not variance_alert.strip():
15368:                 raise ValueError(f"Rule {rule_id} has invalid variance_alert")
15369:             discriminants += 1
15370:
15371:         priority_gaps = when.get('priority_micro_gaps')
15372:         if priority_gaps is not None:
15373:             if not isinstance(priority_gaps, list) or not priority_gaps:
15374:                 raise ValueError(f"Rule {rule_id} must declare non-empty priority_micro_gaps")
15375:             if not all(isinstance(item, str) and item.strip() for item in priority_gaps):
15376:                 raise ValueError(f"Rule {rule_id} has invalid priority_micro_gaps entries")
15377:             discriminants += 1
15378:
15379:         if discriminants == 0:
15380:             raise ValueError(
15381:                 f"Rule {rule_id} must specify at least one MACRO discriminant condition"
15382:             )
15383:
15384:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_template")
15385:     def _validate_template(self, rule_id: str, template: dict[str, Any], level: str) -> None:
15386:         required_fields = ['problem', 'intervention', 'indicator', 'responsible', 'horizon', 'verification', 'template_id', 'template_params']
15387:         for field in required_fields:
15388:             if field not in template:
15389:                 raise ValueError(f"Rule {rule_id} template missing '{field}'")
15390:
15391:         for text_field in ('problem', 'intervention'):
15392:             value = template[text_field]
15393:             if not isinstance(value, str):
15394:                 raise ValueError(f"Rule {rule_id} template field '{text_field}' must be text")
15395:             stripped = value.strip()
15396:             if len(stripped) < 40 or len(stripped.split()) < 12:
15397:                 raise ValueError(
15398:                     f"Rule {rule_id} template field '{text_field}' lacks actionable detail"
15399:                 )
15400:
15401:         indicator = template['indicator']
15402:         if not isinstance(indicator, dict):
15403:             raise ValueError(f"Rule {rule_id} indicator must be an object")

```

```

15404:     for key in ('name', 'target', 'unit'):
15405:         if key not in indicator:
15406:             raise ValueError(f"Rule {rule_id} indicator missing '{key}' field")
15407:
15408:         if not isinstance(indicator['name'], str) or len(indicator['name'].strip()) < 5:
15409:             raise ValueError(f"Rule {rule_id} indicator name too short")
15410:
15411:         target = indicator['target']
15412:         if not self._is_number(target):
15413:             raise ValueError(f"Rule {rule_id} indicator target must be numeric")
15414:
15415:         unit = indicator['unit']
15416:         if not isinstance(unit, str) or not unit.strip():
15417:             raise ValueError(f"Rule {rule_id} indicator unit missing or empty")
15418:
15419:         acceptable_range = indicator.get('acceptable_range')
15420:         if acceptable_range is not None:
15421:             if not isinstance(acceptable_range, list) or len(acceptable_range) != 2:
15422:                 raise ValueError(f"Rule {rule_id} acceptable_range must have two numeric bounds")
15423:             if not all(self._is_number(bound) for bound in acceptable_range):
15424:                 raise ValueError(f"Rule {rule_id} acceptable_range values must be numeric")
15425:             lower, upper = acceptable_range
15426:             if float(lower) >= float(upper):
15427:                 raise ValueError(f"Rule {rule_id} acceptable_range lower bound must be < upper bound")
15428:
15429:         template_id = template['template_id']
15430:         if not isinstance(template_id, str) or not template_id.strip():
15431:             raise ValueError(f"Rule {rule_id} template_id must be a non-empty string")
15432:
15433:         template_params = template['template_params']
15434:         if not isinstance(template_params, dict):
15435:             raise ValueError(f"Rule {rule_id} template_params must be an object")
15436:         allowed_param_keys = {'pa_id', 'dim_id', 'cluster_id', 'question_id'}
15437:         unknown_params = set(template_params) - allowed_param_keys
15438:         if unknown_params:
15439:             raise ValueError(f"Rule {rule_id} template_params contains unsupported keys: {sorted(unknown_params)}")
15440:
15441:         required_params: set[str] = set()
15442:         if level == 'MICRO':
15443:             required_params = {'pa_id', 'dim_id', 'question_id'}
15444:         elif level == 'MESO':
15445:             required_params = {'cluster_id'}
15446:
15447:         missing_params = required_params - set(template_params)
15448:         if missing_params:
15449:             raise ValueError(
15450:                 f"Rule {rule_id} template_params missing required keys for {level}: {sorted(missing_params)}"
15451:             )
15452:
15453:         if level != 'MACRO' and not template_params:
15454:             raise ValueError(f"Rule {rule_id} template_params cannot be empty for {level} level")
15455:
15456:         responsible = template['responsible']
15457:         if not isinstance(responsible, dict):
15458:             raise ValueError(f"Rule {rule_id} responsible must be an object")
15459:         for key in ('entity', 'role'):

```

```
15460:         value = responsible.get(key)
15461:         if not isinstance(value, str) or not value.strip():
15462:             raise ValueError(f"Rule {rule_id} responsible missing '{key}'")
15463:
15464:         partners = responsible.get('partners')
15465:         if partners is None or not isinstance(partners, list) or not partners:
15466:             raise ValueError(f"Rule {rule_id} responsible must enumerate partners")
15467:         if any(not isinstance(partner, str) or not partner.strip() for partner in partners):
15468:             raise ValueError(f"Rule {rule_id} responsible partners must be non-empty strings")
15469:
15470:         horizon = template['horizon']
15471:         if not isinstance(horizon, dict):
15472:             raise ValueError(f"Rule {rule_id} horizon must be an object")
15473:         for key in ('start', 'end'):
15474:             value = horizon.get(key)
15475:             if not isinstance(value, str) or not value.strip():
15476:                 raise ValueError(f"Rule {rule_id} horizon missing '{key}'")
15477:
15478:         verification = template['verification']
15479:         if not isinstance(verification, list) or not verification:
15480:             raise ValueError(f"Rule {rule_id} must define verification artifacts")
15481:         for artifact in verification:
15482:             if not isinstance(artifact, dict):
15483:                 raise ValueError(
15484:                     f"Rule {rule_id} verification entries must be structured dictionaries"
15485:                 )
15486:             required_artifact_fields = (
15487:                 'id',
15488:                 'type',
15489:                 'artifact',
15490:                 'format',
15491:                 'approval_required',
15492:                 'approver',
15493:                 'due_date',
15494:                 'required_sections',
15495:                 'automated_check',
15496:             )
15497:             for key in required_artifact_fields:
15498:                 if key not in artifact:
15499:                     raise ValueError(
15500:                         f"Rule {rule_id} verification artifact missing required field '{key}'"
15501:                     )
15502: # Special handling for boolean fields - they can be False
15503: if key in ('approval_required', 'automated_check'):
15504:     if not isinstance(artifact[key], bool):
15505:         raise ValueError(
15506:             f"Rule {rule_id} verification artifact field '{key}' must be a boolean"
15507:         )
15508: # Special handling for required_sections - must be a list
15509: elif key == 'required_sections':
15510:     if not isinstance(artifact[key], list) or not all(isinstance(s, str) and s.strip() for s in artifact[key]):
15511:         raise ValueError(
15512:             f"Rule {rule_id} verification required_sections must be a list of strings (may be empty)"
15513:         )
15514: # For other non-boolean fields, check for empty values
15515: elif not artifact[key]:
```

```
15516:             raise ValueError(
15517:                 f"Rule {rule_id} verification artifact field '{key}' cannot be empty"
15518:             )
15519:
15520:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_execution")
15521:     def _validate_execution(self, rule_id: str, execution: dict[str, Any]) -> None:
15522:         if not isinstance(execution, dict):
15523:             raise ValueError(f"Rule {rule_id} execution block must be an object")
15524:
15525:         required_keys = {
15526:             'trigger_condition',
15527:             'blocking',
15528:             'auto_apply',
15529:             'requires_approval',
15530:             'approval_roles',
15531:         }
15532:         missing = required_keys - execution.keys()
15533:         if missing:
15534:             raise ValueError(f"Rule {rule_id} execution block missing keys: {sorted(missing)}")
15535:
15536:         if not isinstance(execution['trigger_condition'], str) or not execution['trigger_condition'].strip():
15537:             raise ValueError(f"Rule {rule_id} execution trigger_condition must be a non-empty string")
15538:         for flag in ('blocking', 'auto_apply', 'requires_approval'):
15539:             if not isinstance(execution[flag], bool):
15540:                 raise ValueError(f"Rule {rule_id} execution field '{flag}' must be boolean")
15541:
15542:         roles = execution['approval_roles']
15543:         if not isinstance(roles, list) or not roles:
15544:             raise ValueError(f"Rule {rule_id} execution approval_roles must be a non-empty list")
15545:         if any(not isinstance(role, str) or not role.strip() for role in roles):
15546:             raise ValueError(f"Rule {rule_id} execution approval_roles must contain non-empty strings")
15547:
15548:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_budget")
15549:     def _validate_budget(self, rule_id: str, budget: dict[str, Any]) -> None:
15550:         if not isinstance(budget, dict):
15551:             raise ValueError(f"Rule {rule_id} budget block must be an object")
15552:
15553:         required_keys = {'estimated_cost_cop', 'cost_breakdown', 'funding_sources', 'fiscal_year'}
15554:         missing = required_keys - budget.keys()
15555:         if missing:
15556:             raise ValueError(f"Rule {rule_id} budget block missing keys: {sorted(missing)}")
15557:
15558:         if not self._is_number(budget['estimated_cost_cop']):
15559:             raise ValueError(f"Rule {rule_id} budget estimated_cost_cop must be numeric")
15560:
15561:         cost_breakdown = budget['cost_breakdown']
15562:         if not isinstance(cost_breakdown, dict) or not cost_breakdown:
15563:             raise ValueError(f"Rule {rule_id} cost_breakdown must be a non-empty object")
15564:         for key, value in cost_breakdown.items():
15565:             if not isinstance(key, str) or not key.strip():
15566:                 raise ValueError(f"Rule {rule_id} cost_breakdown keys must be non-empty strings")
15567:             if not self._is_number(value):
15568:                 raise ValueError(f"Rule {rule_id} cost_breakdown values must be numeric")
15569:
15570:         funding_sources = budget['funding_sources']
15571:         if not isinstance(funding_sources, list) or not funding_sources:
```

```

15572:         raise ValueError(f"Rule {rule_id} funding_sources must be a non-empty list")
15573:     for source in funding_sources:
15574:         if not isinstance(source, dict):
15575:             raise ValueError(f"Rule {rule_id} funding source entries must be objects")
15576:         for key in ('source', 'amount', 'confirmed'):
15577:             if key not in source:
15578:                 raise ValueError(f"Rule {rule_id} funding source missing '{key}'")
15579:             if not isinstance(source['source'], str) or not source['source'].strip():
15580:                 raise ValueError(f"Rule {rule_id} funding source name must be a non-empty string")
15581:             if not self._is_number(source['amount']):
15582:                 raise ValueError(f"Rule {rule_id} funding source amount must be numeric")
15583:             if not isinstance(source['confirmed'], bool):
15584:                 raise ValueError(f"Rule {rule_id} funding source confirmed flag must be boolean")
15585:
15586:     fiscal_year = budget['fiscal_year']
15587:     if not isinstance(fiscal_year, int):
15588:         raise ValueError(f"Rule {rule_id} fiscal_year must be an integer")
15589:
15590:     @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._validate_ruleset_metadata")
15591:     def _validate_ruleset_metadata(self) -> None:
15592:         version = self.rules.get('version')
15593:         if not isinstance(version, str) or not version.startswith('2.0'):
15594:             raise ValueError(
15595:                 "Enhanced recommendation engine requires ruleset version 2.0"
15596:             )
15597:
15598:         features = self.rules.get('enhanced_features')
15599:         if not isinstance(features, list) or not features:
15600:             raise ValueError("Enhanced recommendation engine requires enhanced_features list")
15601:
15602:         feature_set = {feature for feature in features if isinstance(feature, str)}
15603:         missing = _REQUIRED_ENHANCED_FEATURES - feature_set
15604:         if missing:
15605:             raise ValueError(
15606:                 f"Enhanced recommendation rules missing required features: {sorted(missing)}"
15607:             )
15608:
15609:     @staticmethod
15610:     def _is_number(value: Any) -> bool:
15611:         return isinstance(value, (int, float)) and not isinstance(value, bool)
15612:
15613:     def generate_all_recommendations(
15614:         self,
15615:         micro_scores: dict[str, float],
15616:         cluster_data: dict[str, Any],
15617:         macro_data: dict[str, Any],
15618:         context: dict[str, Any] | None = None
15619:     ) -> dict[str, RecommendationSet]:
15620:         """
15621:             Generate recommendations at all three levels
15622:
15623:             Args:
15624:                 micro_scores: PA-DIM scores for MICRO recommendations
15625:                 cluster_data: Cluster metrics for MESO recommendations
15626:                 macro_data: Plan-level metrics for MACRO recommendations
15627:                 context: Additional context

```

```
15628:  
15629:     Returns:  
15630:         Dictionary with 'MICRO', 'MESO', and 'MACRO' recommendation sets  
15631:     """  
15632:     return {  
15633:         'MICRO': self.generate_micro_recommendations(micro_scores, context),  
15634:         'MESO': self.generate_meso_recommendations(cluster_data, context),  
15635:         'MACRO': self.generate_macro_recommendations(macro_data, context)  
15636:     }  
15637:  
15638:     def export_recommendations(  
15639:         self,  
15640:         recommendations: dict[str, RecommendationSet],  
15641:         output_path: str,  
15642:         format: str = 'json'  
15643:     ) -> None:  
15644:     """  
15645:         Export recommendations to file  
15646:  
15647:         Args:  
15648:             recommendations: Dictionary of recommendation sets  
15649:             output_path: Path to output file  
15650:             format: Output format ('json' or 'markdown')  
15651:         """  
15652:         # Delegate to factory for I/O operation  
15653:         from farfan_pipeline.analysis.factory import save_json, write_text_file  
15654:  
15655:         if format == 'json':  
15656:             save_json(  
15657:                 {level: rec_set.to_dict() for level, rec_set in recommendations.items()},  
15658:                 output_path  
15659:             )  
15660:         elif format == 'markdown':  
15661:             write_text_file(  
15662:                 self._format_as_markdown(recommendations),  
15663:                 output_path  
15664:             )  
15665:         else:  
15666:             raise ValueError(f"Unsupported format: {format}")  
15667:  
15668:             logger.info(f"Exported recommendations to {output_path} in {format} format")  
15669:  
15670: @calibrated_method("farfan_core.analysis.recommendation_engine.RecommendationEngine._format_as_markdown")  
15671: def _format_as_markdown(self, recommendations: dict[str, RecommendationSet]) -> str:  
15672:     """Format recommendations as Markdown"""  
15673:     lines = ["# Recomendaciones del Plan de Desarrollo\n"]  
15674:  
15675:     for level in ['MICRO', 'MESO', 'MACRO']:  
15676:         rec_set = recommendations.get(level)  
15677:         if not rec_set:  
15678:             continue  
15679:  
15680:             lines.append(f"\n## Nivel {level}\n")  
15681:             lines.append(f"**Generado:** {rec_set.generated_at}\n")  
15682:             lines.append(f"**Reglas evaluadas:** {rec_set.total_rules_evaluated}\n")  
15683:             lines.append(f"**Recomendaciones:** {rec_set.rules_matched}\n")
```

```

15684:
15685:         for i, rec in enumerate(rec_set.recommendations, 1):
15686:             lines.append(f"\n### {i}. {rec.rule_id}\n")
15687:             lines.append(f"**Problema:** {rec.problem}\n")
15688:             lines.append(f"\n**IntervenciÃ³n:** {rec.intervention}\n")
15689:             lines.append("\n**Indicador:**")
15690:             lines.append(f"- Nombre: {rec.indicator.get('name')}")
15691:             lines.append(f"- Meta: {rec.indicator.get('target')} {rec.indicator.get('unit')}\n")
15692:             lines.append(f"\n**Responsable:** {rec.responsible.get('entity')} ({rec.responsible.get('role')})\n")
15693:             lines.append(f"\n**Socios:** {', '.join(rec.responsible.get('partners', []))}\n")
15694:             lines.append(f"\n**Horizonte:** {rec.horizon.get('start')} \u2013 {rec.horizon.get('end')}\n")
15695:             lines.append("\n**VerificaciÃ³n:**")
15696:
15697:             for v in rec.verification:
15698:                 if isinstance(v, dict):
15699:                     descriptor = f"[{v.get('type', 'ARTIFACT')}] {v.get('artifact', 'Sin artefacto')}"
15700:                     due = v.get('due_date')
15701:                     approver = v.get('approver')
15702:                     suffix_parts: list[str] = []
15703:                     if due:
15704:                         suffix_parts.append(f"entrega: {due}")
15705:                     if approver:
15706:                         suffix_parts.append(f"aprueba: {approver}")
15707:                     suffix = f" ({'; '.join(suffix_parts)})" if suffix_parts else ""
15708:                     lines.append(f"- {descriptor}{suffix}")
15709:                     sections = v.get('required_sections') or []
15710:                     if sections:
15711:                         lines.append(
15712:                             " - Secciones requeridas: " + ", ".join(str(section) for section in sections)
15713:                         )
15714:                     else:
15715:                         lines.append(f"- {v}")
15716:
15717:             return "\n".join(lines)
15718:
15719: # =====
15720: # CONVENIENCE FUNCTIONS
15721: # =====
15722:
15723: def load_recommendation_engine(
15724:     rules_path: str = "config/recommendation_rules_enhanced.json",
15725:     schema_path: str = "rules/recommendation_rules.schema.json"
15726: ) -> RecommendationEngine:
15727:     """
15728:     Convenience function to load recommendation engine
15729:
15730:     Args:
15731:         rules_path: Path to rules JSON
15732:         schema_path: Path to schema JSON
15733:
15734:     Returns:
15735:         Initialized RecommendationEngine
15736:     """
15737:     return RecommendationEngine(rules_path=rules_path, schema_path=schema_path)
15738:
15739: # Note: Main entry point removed to maintain I/O boundary separation.

```

```
15740: # For usage examples, see examples/ directory.
15741:
15742:
15743:
15744: =====
15745: FILE: src/farfan_pipeline/analysis/report_assembly.py
15746: =====
15747:
15748: """
15749: Report Assembly Module - Production Grade v2.0
15750: =====
15751:
15752: This module assembles comprehensive policy analysis reports by:
15753: 1. Loading questionnaire monolith via factory (I/O boundary)
15754: 2. Accessing patterns via QuestionnaireResourceProvider (single source of truth)
15755: 3. Integrating with evidence registry and QMCM hooks
15756: 4. Producing structured, traceable reports with cryptographic verification
15757:
15758: Architectural Compliance:
15759: - REQUIREMENT 1: Uses QuestionnaireResourceProvider for pattern extraction
15760: - REQUIREMENT 2: All I/O via factory.py
15761: - REQUIREMENT 3: Receives dependencies via dependency injection
15762: - REQUIREMENT 4: Domain-specific exceptions with structured payloads
15763: - REQUIREMENT 5: Pydantic contracts for data validation
15764: - REQUIREMENT 6: Cryptographic verification (SHA-256)
15765: - REQUIREMENT 7: Structured JSON logging
15766: - REQUIREMENT 8: Parameter externalization via calibration system
15767:
15768: Author: Integration Team
15769: Version: 2.0.0
15770: Python: 3.10+
15771: """
15772:
15773: from __future__ import annotations
15774:
15775: import hashlib
15776: import json
15777: import logging
15778: import uuid
15779: from datetime import datetime, timezone
15780: from typing import TYPE_CHECKING, Any
15781:
15782: from pydantic import BaseModel, ConfigDict, Field, field_validator
15783: # Calibration parameters - loaded at runtime if calibration system available
15784: try:
15785:     from farfan_pipeline.core.parameters import ParameterLoaderV2
15786: except (ImportError, AttributeError):
15787:     # Fallback: use explicit defaults if calibration system not available
15788:     _PARAM_LOADER = None
15789:
15790: # Calibrated method decorator stub (calibration system not available)
15791: def calibrated_method(method_name: str):
15792:     """No-op decorator stub for compatibility when calibration system unavailable."""
15793:     def decorator(func):
15794:         return func
15795:     return decorator
```

```
15796:  
15797: if TYPE_CHECKING:  
15798:     from pathlib import Path  
15799:  
15800: logger = logging.getLogger(__name__)  
15801:  
15802: # =====  
15803: # DOMAIN-SPECIFIC EXCEPTIONS  
15804: # =====  
15805:  
15806: class ReportAssemblyException(Exception):  
15807:     """Base exception for report assembly operations with structured payloads."""  
15808:  
15809:     def __init__(  
15810:         self,  
15811:             message: str,  
15812:             details: dict[str, Any] | None = None,  
15813:             stage: str | None = None,  
15814:             recoverable: bool = False,  
15815:             event_id: str | None = None  
15816:     ) -> None:  
15817:         self.message = message  
15818:         self.details = details or {}  
15819:         self.stage = stage  
15820:         self.recoverable = recoverable  
15821:         self.event_id = event_id or str(uuid.uuid4())  
15822:         super().__init__(self._format_message())  
15823:  
15824:     def _format_message(self) -> str:  
15825:         """Format error message with structured information."""  
15826:         parts = ["[ReportAssembly Error]"]  
15827:         if self.stage:  
15828:             parts.append(f"[Stage: {self.stage}]")  
15829:         parts.append(f"[EventID: {self.event_id[:8]}]")  
15830:         parts.append(self.message)  
15831:         if self.details:  
15832:             parts.append(f"Details: {json.dumps(self.details, indent=2)}")  
15833:         return " ".join(parts)  
15834:  
15835:     def to_dict(self) -> dict[str, Any]:  
15836:         """Convert exception to structured dictionary."""  
15837:         return {  
15838:             'error_type': self.__class__.__name__,  
15839:             'message': self.message,  
15840:             'details': self.details,  
15841:             'stage': self.stage,  
15842:             'recoverable': self.recoverable,  
15843:             'event_id': self.event_id  
15844:         }  
15845:  
15846:  
15847: class ReportValidationError(ReportAssemblyException):  
15848:     """Raised when report data validation fails."""  
15849:     pass  
15850:  
15851:
```

```
15852: class ReportIntegrityError(ReportAssemblyException):
15853:     """Raised when cryptographic verification fails (hash mismatch)."""
15854:     pass
15855:
15856:
15857: class ReportExportError(ReportAssemblyException):
15858:     """Raised when report export to file fails."""
15859:     pass
15860:
15861:
15862: # =====
15863: # UTILITY FUNCTIONS
15864: # =====
15865:
15866: def compute_content_digest(content: str | bytes | dict[str, Any]) -> str:
15867:     """
15868:         Compute SHA-256 digest of content in a deterministic way.
15869:
15870:     Args:
15871:         content: String, bytes, or dict to hash
15872:
15873:     Returns:
15874:         Hexadecimal SHA-256 digest (64 characters)
15875:
15876:     Raises:
15877:         ReportValidationError: If content type is unsupported
15878:     """
15879:     if isinstance(content, dict):
15880:         # Sort keys for deterministic JSON
15881:         content_str = json.dumps(content, sort_keys=True, ensure_ascii=True, separators=(',', ':'))
15882:         content_bytes = content_str.encode('utf-8')
15883:     elif isinstance(content, str):
15884:         content_bytes = content.encode('utf-8')
15885:     elif isinstance(content, bytes):
15886:         content_bytes = content
15887:     else:
15888:         raise ReportValidationError(
15889:             f"Cannot compute digest for type {type(content).__name__}",
15890:             details={'content_type': type(content).__name__},
15891:             stage="digest_computation"
15892:         )
15893:
15894:     return hashlib.sha256(content_bytes).hexdigest()
15895:
15896:
15897: def utc_now_iso() -> str:
15898:     """
15899:         Get current UTC timestamp in ISO-8601 format.
15900:
15901:     Returns:
15902:         ISO-8601 timestamp string (UTC timezone)
15903:     """
15904:     return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')
15905:
15906:
15907: # =====
```

```
15908: # PYDANTIC CONTRACT MODELS
15909: # =====
15910:
15911: class ReportMetadata(BaseModel):
15912:     """Enhanced metadata for analysis report with cryptographic traceability."""
15913:
15914:     model_config = ConfigDict(
15915:         frozen=True,
15916:         extra='forbid',
15917:         validate_assignment=True,
15918:         str_strip_whitespace=True,
15919:     )
15920:
15921:     report_id: str = Field(..., description="Unique report identifier", min_length=1)
15922:     generated_at: str = Field(
15923:         default_factory=utc_now_iso,
15924:         description="UTC timestamp in ISO-8601 format"
15925:     )
15926:     monolith_version: str = Field(..., description="Questionnaire monolith version")
15927:     monolith_hash: str = Field(
15928:         ...,
15929:         description="SHA-256 hash of questionnaire_monolith.json",
15930:         pattern=r"^[a-f0-9]{64}$"
15931:     )
15932:     plan_name: str = Field(..., description="Development plan name", min_length=1)
15933:     total_questions: int = Field(..., description="Total number of questions", ge=0)
15934:     questions_analyzed: int = Field(..., description="Number of questions analyzed", ge=0)
15935:     metadata: dict[str, Any] = Field(default_factory=dict, description="Additional metadata")
15936:     correlation_id: str = Field(
15937:         default_factory=lambda: str(uuid.uuid4()),
15938:         description="UUID for request correlation"
15939:     )
15940:
15941:     @field_validator('generated_at')
15942:     @classmethod
15943:     def validate_timestamp(cls, v: str) -> str:
15944:         """Validate timestamp is ISO-8601 format and UTC."""
15945:         try:
15946:             dt = datetime.fromisoformat(v.replace('Z', '+00:00'))
15947:             # Ensure UTC
15948:             if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
15949:                 raise ValueError("Timestamp must be UTC")
15950:             return v
15951:         except (ValueError, AttributeError) as e:
15952:             raise ReportValidationError(
15953:                 f"Invalid ISO-8601 timestamp: {v}",
15954:                 details={'timestamp': v, 'error': str(e)},
15955:                 stage="metadata_validation"
15956:             ) from e
15957:
15958:     @field_validator('questions_analyzed')
15959:     @classmethod
15960:     def validate_analyzed_count(cls, v: int, info) -> int:
15961:         """Validate analyzed count doesn't exceed total."""
15962:         # Note: 'total_questions' may not be available yet during construction
15963:         # This is validated in post_init if needed
```

```
15964:         if v < 0:
15965:             raise ReportValidationError(
15966:                 "questions_analyzed must be non-negative",
15967:                 details={'questions_analyzed': v},
15968:                 stage="metadata_validation"
15969:             )
15970:     return v
15971:
15972:
15973: class QuestionAnalysis(BaseModel):
15974:     """Enhanced analysis result for a single micro question."""
15975:
15976:     model_config = ConfigDict(
15977:         frozen=True,
15978:         extra='forbid',
15979:         validate_assignment=True,
15980:     )
15981:
15982:     question_id: str = Field(..., description="Question identifier", min_length=1)
15983:     question_global: int = Field(..., description="Global question number", ge=1, le=500)
15984:     base_slot: str = Field(..., description="Base slot identifier")
15985:     scoring_modality: str | None = Field(default=None, description="Scoring modality")
15986:     score: float | None = Field(default=None, description="Question score", ge=0.0, le=1.0)
15987:     evidence: list[str] = Field(default_factory=list, description="Evidence list")
15988:     patterns_applied: list[str] = Field(default_factory=list, description="Applied pattern IDs")
15989:     recommendation: str | None = Field(default=None, description="Analysis recommendation")
15990:     metadata: dict[str, Any] = Field(default_factory=dict, description="Additional metadata")
15991:
15992:     @field_validator('score')
15993:     @classmethod
15994:     def validate_score_bounds(cls, v: float | None) -> float | None:
15995:         """Validate score is within bounds if present."""
15996:         if v is not None:
15997:             min_score = 0.0
15998:             max_score = 1.0
15999:             if not (min_score <= v <= max_score):
16000:                 raise ReportValidationError(
16001:                     f"Score must be in [{min_score}, {max_score}], got {v}",
16002:                     details={'score': v, 'min': min_score, 'max': max_score},
16003:                     stage="question_validation"
16004:                 )
16005:             # Round to avoid floating point precision issues
16006:             return round(v, 6)
16007:             return round(v, 6)
16008:         return v
16009:
16010:
16011: class Recommendation(BaseModel):
16012:     """Structured recommendation with type and severity classification."""
16013:
16014:     model_config = ConfigDict(frozen=True, extra='forbid')
16015:
16016:     type: str = Field(..., description="Recommendation type (RISK, PRIORITY, OMISSION, etc.)")
16017:     severity: str = Field(..., description="Severity level (CRITICAL, HIGH, MEDIUM, LOW, INFO)")
16018:     description: str = Field(..., description="Actionable recommendation text")
16019:     source: str = Field(default="macro", description="Source of recommendation (micro, meso, macro)")
```

```
16020:  
16021:     @classmethod  
16022:     def from_string(cls, text: str, source: str = "macro") -> "Recommendation":  
16023:         """Parse recommendation string into structured object."""  
16024:         # Expected format: "TYPE_LEVEL: Description"  
16025:         # e.g., "CRITICAL_RISK: Immediate intervention required"  
16026:         if ":" in text:  
16027:             prefix, desc = text.split(":", 1)  
16028:             desc = desc.strip()  
16029:  
16030:             # Parse prefix like "CRITICAL_RISK" -> severity="CRITICAL", type="RISK"  
16031:             parts = prefix.split("_")  
16032:             if len(parts) >= 2:  
16033:                 severity = parts[0]  
16034:                 rec_type = "_".join(parts[1:])  
16035:             else:  
16036:                 severity = "INFO"  
16037:                 rec_type = prefix  
16038:             else:  
16039:                 severity = "INFO"  
16040:                 rec_type = "GENERAL"  
16041:             desc = text  
16042:  
16043:             return cls(  
16044:                 type=rec_type,  
16045:                 severity=severity,  
16046:                 description=desc,  
16047:                 source=source  
16048:             )  
16049:  
16050:  
16051: class MesoCluster(BaseModel):  
16052:     """Validated meso-level cluster analysis."""  
16053:  
16054:     model_config = ConfigDict(frozen=True, extra='forbid')  
16055:  
16056:     cluster_id: str = Field(..., min_length=1)  
16057:     raw_meso_score: float = Field(..., ge=0.0, le=1.0)  
16058:     adjusted_score: float = Field(..., ge=0.0, le=1.0)  
16059:  
16060:     # Penalties  
16061:     dispersion_penalty: float = Field(..., ge=0.0, le=1.0)  
16062:     peer_penalty: float = Field(..., ge=0.0, le=1.0)  
16063:     total_penalty: float = Field(..., ge=0.0, le=1.0)  
16064:  
16065:     # Metrics  
16066:     dispersion_metrics: dict[str, float] = Field(default_factory=dict)  
16067:     micro_scores: list[float] = Field(default_factory=list)  
16068:  
16069:     metadata: dict[str, Any] = Field(default_factory=dict)  
16070:  
16071:  
16072: class MacroSummary(BaseModel):  
16073:     """Validated macro-level portfolio analysis."""  
16074:  
16075:     model_config = ConfigDict(frozen=True, extra='forbid')
```

```
16076:
16077:     overall_posterior: float = Field(..., ge=0.0, le=1.0)
16078:     adjusted_score: float = Field(..., ge=0.0, le=1.0)
16079:
16080:     # Penalties
16081:     coverage_penalty: float = Field(..., ge=0.0, le=1.0)
16082:     dispersion_penalty: float = Field(..., ge=0.0, le=1.0)
16083:     contradiction_penalty: float = Field(..., ge=0.0, le=1.0)
16084:     total_penalty: float = Field(..., ge=0.0, le=1.0)
16085:
16086:     # Counts
16087:     contradiction_count: int = Field(..., ge=0)
16088:
16089:     # Recommendations
16090:     recommendations: list[Recommendation] = Field(default_factory=list)
16091:
16092:     metadata: dict[str, Any] = Field(default_factory=dict)
16093:
16094:
16095: class AnalysisReport(BaseModel):
16096:     """Enhanced complete policy analysis report with cryptographic verification."""
16097:
16098:     model_config = ConfigDict(
16099:         frozen=True,
16100:         extra='forbid',
16101:         validate_assignment=True,
16102:     )
16103:
16104:     metadata: ReportMetadata = Field(..., description="Report metadata")
16105:     micro_analyses: list[QuestionAnalysis] = Field(..., description="Micro-level analyses")
16106:     meso_clusters: dict[str, MesoCluster] = Field(default_factory=dict, description="Meso-level clusters")
16107:     macro_summary: MacroSummary | None = Field(default=None, description="Macro-level summary")
16108:     evidence_chain_hash: str | None = Field(
16109:         default=None,
16110:         description="Evidence chain hash",
16111:         pattern=r"^[a-f0-9]{64}$"
16112:     )
16113:     report_digest: str | None = Field(
16114:         default=None,
16115:         description="SHA-256 digest of report content",
16116:         pattern=r"^[a-f0-9]{64}$"
16117:     )
16118:
16119:     @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.to_dict")
16120:     def to_dict(self) -> dict[str, Any]:
16121:         """Convert report to dictionary for JSON serialization."""
16122:         report_dict = {
16123:             'metadata': self.metadata.model_dump(),
16124:             'micro_analyses': [q.model_dump() for q in self.micro_analyses],
16125:             'meso_clusters': {k: v.model_dump() for k, v in self.meso_clusters.items()},
16126:             'macro_summary': self.macro_summary.model_dump() if self.macro_summary else None,
16127:             'evidence_chain_hash': self.evidence_chain_hash,
16128:             'report_digest': self.report_digest
16129:         }
16130:         return report_dict
16131:
```

```
16132:     @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.compute_digest")
16133:     def compute_digest(self) -> str:
16134:         """Compute cryptographic digest of report content."""
16135:         # Create deterministic representation without the digest field
16136:         content = {
16137:             'metadata': self.metadata.model_dump(),
16138:             'micro_analyses': [q.model_dump() for q in self.micro_analyses],
16139:             'meso_clusters': {k: v.model_dump() for k, v in self.meso_clusters.items()},
16140:             'macro_summary': self.macro_summary.model_dump() if self.macro_summary else None,
16141:             'evidence_chain_hash': self.evidence_chain_hash
16142:         }
16143:         return compute_content_digest(content)
16144:
16145:     @calibrated_method("farfan_core.analysis.report_assembly.AnalysisReport.verify_digest")
16146:     def verify_digest(self) -> bool:
16147:         """Verify report digest matches computed hash."""
16148:         if self.report_digest is None:
16149:             return False
16150:         computed = self.compute_digest()
16151:         return computed == self.report_digest
16152:
16153:
16154: # =====
16155: # STRUCTURED LOGGING HELPER
16156: # =====
16157:
16158: class ReportLogger:
16159:     """Structured JSON logger for report assembly operations."""
16160:
16161:     def __init__(self, name: str) -> None:
16162:         """Initialize logger with name."""
16163:         self.logger = logging.getLogger(name)
16164:         self.logger.setLevel(logging.INFO)
16165:
16166:     def log_operation(
16167:         self,
16168:         operation: str,
16169:         correlation_id: str,
16170:         success: bool,
16171:         latency_ms: float,
16172:         **kwargs: Any
16173:     ) -> None:
16174:         """Log operation event with structured data."""
16175:         log_entry = {
16176:             "event": "report_operation",
16177:             "operation": operation,
16178:             "correlation_id": correlation_id,
16179:             "success": success,
16180:             "latency_ms": round(latency_ms, 3),
16181:             "timestamp_utc": utc_now_iso(),
16182:         }
16183:         log_entry.update(kwargs)
16184:
16185:         self.logger.info(json.dumps(log_entry, sort_keys=True))
16186:
16187:     def log_validation(
```

```
16188:         self,
16189:         item_type: str,
16190:         correlation_id: str,
16191:         success: bool,
16192:         error: str | None = None,
16193:         **kwargs: Any
16194:     ) -> None:
16195:         """Log validation event."""
16196:         log_entry = {
16197:             "event": "report_validation",
16198:             "item_type": item_type,
16199:             "correlation_id": correlation_id,
16200:             "success": success,
16201:             "timestamp_utc": utc_now_iso(),
16202:         }
16203:         if error:
16204:             log_entry["error"] = error
16205:         log_entry.update(kwargs)
16206:
16207:         self.logger.info(json.dumps(log_entry, sort_keys=True))
16208:
16209:
16210: # =====
16211: # REPORT ASSEMBLER
16212: # =====
16213:
16214: class ReportAssembler:
16215:     """
16216:     Assembles comprehensive policy analysis reports.
16217:
16218:     This class demonstrates proper architectural patterns:
16219:     - Dependency injection for all external resources
16220:     - No direct file I/O (delegates to factory)
16221:     - Pattern extraction via QuestionnaireResourceProvider
16222:     - Cryptographic traceability via SHA-256 digests
16223:     - Domain-specific exceptions with structured payloads
16224:     - Pydantic contract validation
16225:     - Structured JSON logging
16226:     """
16227:
16228:     def __init__(
16229:         self,
16230:         questionnaire_provider,
16231:         evidence_registry=None,
16232:         qmcm_recorder=None,
16233:         orchestrator=None
16234:     ) -> None:
16235:         """
16236:         Initialize report assembler.
16237:
16238:         Args:
16239:             questionnaire_provider: QuestionnaireResourceProvider instance (required)
16240:             evidence_registry: EvidenceRegistry for traceability (optional)
16241:             qmcm_recorder: QMCMRecorder for quality monitoring (optional)
16242:             orchestrator: Orchestrator instance for execution results (optional)
16243:
```

```
16244:     ARCHITECTURAL NOTE: All dependencies injected, no direct I/O.
16245:     """
16246:     if questionnaire_provider is None:
16247:         raise ReportValidationError(
16248:             "questionnaire_provider is required",
16249:             details={'provider': None},
16250:             stage="initialization",
16251:             recoverable=False
16252:         )
16253:
16254:     self.questionnaire_provider = questionnaire_provider
16255:     self.evidence_registry = evidence_registry
16256:     self.qmcm_recorder = qmcm_recorder
16257:     self.orchestrator = orchestrator
16258:     self.report_logger = ReportLogger(__name__)
16259:
16260:     logger.info("ReportAssembler initialized with dependency injection")
16261:
16262:     @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler.assemble_report")
16263:     def assemble_report(
16264:         self,
16265:         plan_name: str,
16266:         execution_results: dict[str, Any],
16267:         report_id: str | None = None,
16268:         enriched_packs: dict[str, Any] | None = None
16269:     ) -> AnalysisReport:
16270:         """
16271:             Assemble complete analysis report.
16272:
16273:             Args:
16274:                 plan_name: Name of the development plan
16275:                 execution_results: Results from orchestrator execution
16276:                 report_id: Optional report identifier
16277:
16278:             Returns:
16279:                 Structured AnalysisReport with full traceability
16280:
16281:             Raises:
16282:                 ReportValidationError: If input validation fails
16283:                 ReportIntegrityError: If hash computation fails
16284:             """
16285:     import time
16286:     start_time = time.time()
16287:
16288:     # Input validation
16289:     if not plan_name or not isinstance(plan_name, str):
16290:         raise ReportValidationError(
16291:             "plan_name must be a non-empty string",
16292:             details={'plan_name': plan_name, 'type': type(plan_name).__name__},
16293:             stage="input_validation"
16294:         )
16295:
16296:     if not isinstance(execution_results, dict):
16297:         raise ReportValidationError(
16298:             "execution_results must be a dictionary",
16299:             details={'type': type(execution_results).__name__},
```

```
16300:             stage="input_validation"
16301:         )
16302:
16303:     # Generate report ID if not provided
16304:     if report_id is None:
16305:         timestamp = datetime.now(timezone.utc).strftime("%Y%m%d_%H%M%S")
16306:         report_id = f"report_{plan_name}_{timestamp}"
16307:
16308:     correlation_id = str(uuid.uuid4())
16309:
16310:     try:
16311:         # Get questionnaire data and compute hash
16312:         questionnaire_data = self.questionnaire_provider.get_data()
16313:
16314:         if not isinstance(questionnaire_data, dict):
16315:             raise ReportIntegrityError(
16316:                 "Invalid questionnaire data format",
16317:                 details={'type': type(questionnaire_data).__name__},
16318:                 stage="questionnaire_loading"
16319:             )
16320:
16321:         # Import hash utility for content verification
16322:         from farfan_pipeline.utils.hash_utils import compute_hash
16323:         monolith_hash = compute_hash(questionnaire_data)
16324:
16325:         # Validate hash format
16326:         if not isinstance(monolith_hash, str) or len(monolith_hash) != 64:
16327:             raise ReportIntegrityError(
16328:                 "Invalid monolith hash format",
16329:                 details={'hash': monolith_hash, 'length': len(monolith_hash) if isinstance(monolith_hash, str) else 0},
16330:                 stage="hash_computation"
16331:             )
16332:
16333:         # Extract metadata with defensive checks
16334:         version = questionnaire_data.get('version', 'unknown')
16335:         blocks = questionnaire_data.get('blocks', {})
16336:
16337:         if not isinstance(blocks, dict):
16338:             raise ReportValidationError(
16339:                 "questionnaire blocks must be a dictionary",
16340:                 details={'type': type(blocks).__name__},
16341:                 stage="data_extraction"
16342:             )
16343:
16344:         micro_questions = blocks.get('micro_questions', [])
16345:
16346:         if not isinstance(micro_questions, list):
16347:             raise ReportValidationError(
16348:                 "micro_questions must be a list",
16349:                 details={'type': type(micro_questions).__name__},
16350:                 stage="data_extraction"
16351:
16352:         # Create report metadata with Pydantic validation
16353:         metadata = ReportMetadata(
16354:             report_id=report_id,
16355:             generated_at=utc_now_iso(),
```

```
16356:             monolith_hash=monolith_hash,
16357:             plan_name=plan_name,
16358:             total_questions=len(micro_questions),
16359:             questions_analyzed=len(execution_results.get('questions', {})),
16360:             correlation_id=correlation_id
16361:         )
16362:
16363:     # Assemble micro analyses
16364:     micro_analyses = self._assemble_micro_analyses(
16365:         micro_questions,
16366:         execution_results,
16367:         correlation_id
16368:     )
16369:
16370:     # Assemble meso clusters
16371:     meso_clusters = self._assemble_meso_clusters(execution_results)
16372:
16373:     # Assemble macro summary
16374:     macro_summary = self._assemble_macro_summary(execution_results)
16375:
16376:     # Get evidence chain hash if available
16377:     evidence_chain_hash = None
16378:     if self.evidence_registry is not None:
16379:         records = self.evidence_registry.records
16380:         if records:
16381:             evidence_chain_hash = records[-1].entry_hash
16382:
16383:     # JOBFRONT 9: Compute signal usage summary if enriched_packs provided
16384:     if enriched_packs:
16385:         signal_usage = self._compute_signal_usage_summary(
16386:             execution_results,
16387:             enriched_packs
16388:         )
16389:         # Add to metadata
16390:         if metadata.metadata is None:
16391:             # metadata.metadata is immutable, need to recreate
16392:             from dataclasses import replace
16393:             new_metadata_dict = {
16394:                 'signal_version': '1.0.0',
16395:                 'total_patterns_available': signal_usage['total_patterns_available'],
16396:                 'total_patterns_used': signal_usage['total_patterns_used'],
16397:                 'signal_usage_summary': signal_usage
16398:             }
16399:             metadata = ReportMetadata(
16400:                 report_id=metadata.report_id,
16401:                 generated_at=metadata.generated_at,
16402:                 monolith_version=metadata.monolith_version,
16403:                 monolith_hash=metadata.monolith_hash,
16404:                 plan_name=metadata.plan_name,
16405:                 total_questions=metadata.total_questions,
16406:                 questions_analyzed=metadata.questions_analyzed,
16407:                 metadata=new_metadata_dict,
16408:                 correlation_id=metadata.correlation_id
16409:             )
16410:
16411:     # Create report and compute digest
```

```
16412:         report = AnalysisReport(
16413:             metadata=metadata,
16414:             micro_analyses=micro_analyses,
16415:             meso_clusters=meso_clusters,
16416:             macro_summary=macro_summary,
16417:             evidence_chain_hash=evidence_chain_hash,
16418:             report_digest=None # Will be computed
16419:         )
16420:
16421:         # Compute and attach digest
16422:         report_digest = report.compute_digest()
16423:         report = AnalysisReport(
16424:             metadata=metadata,
16425:             micro_analyses=micro_analyses,
16426:             meso_clusters=meso_clusters,
16427:             macro_summary=macro_summary,
16428:             evidence_chain_hash=evidence_chain_hash,
16429:             report_digest=report_digest
16430:         )
16431:
16432:         latency_ms = (time.time() - start_time) * 1000
16433:
16434:         # Structured logging
16435:         self.report_logger.log_operation(
16436:             operation="assemble_report",
16437:             correlation_id=correlation_id,
16438:             success=True,
16439:             latency_ms=latency_ms,
16440:             report_id=report_id,
16441:             question_count=len(micro_analyses),
16442:             monolith_hash=monolith_hash[:16],
16443:             report_digest=report_digest[:16]
16444:         )
16445:
16446:         logger.info(
16447:             f"Report assembled: {report_id} "
16448:             f"({len(micro_analyses)} questions, hash: {monolith_hash[:16]}...)"
16449:         )
16450:
16451:         return report
16452:
16453:     except ReportAssemblyException:
16454:         # Re-raise our domain exceptions
16455:         raise
16456:     except Exception as e:
16457:         # Wrap unexpected exceptions
16458:         raise ReportAssemblyException(
16459:             f"Unexpected error during report assembly: {str(e)}",
16460:             details={'error_type': type(e).__name__, 'error': str(e)},
16461:             stage="assembly",
16462:             recoverable=False
16463:         ) from e
16464:
16465:     @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_micro_analyses")
16466:     def _assemble_micro_analyses(
16467:         self,
```

```
16468:     micro_questions: list[dict[str, Any]],
16469:     execution_results: dict[str, Any],
16470:     correlation_id: str
16471: ) -> list[QuestionAnalysis]:
16472:     """Assemble micro-level question analyses with validation."""
16473:     analyses = []
16474:     question_results = execution_results.get('questions', {})
16475:
16476:     if not isinstance(question_results, dict):
16477:         raise ReportValidationError(
16478:             "execution_results.questions must be a dictionary",
16479:             details={'type': type(question_results).__name__},
16480:             stage="micro_analysis"
16481:         )
16482:
16483:     for question in micro_questions:
16484:         if not isinstance(question, dict):
16485:             logger.warning(f"Skipping invalid question entry: {type(question).__name__}")
16486:             continue
16487:
16488:         question_id = question.get('question_id', '')
16489:         if not question_id:
16490:             logger.warning("Skipping question with missing question_id")
16491:             continue
16492:
16493:         result = question_results.get(question_id, {})
16494:
16495:         # Extract patterns applied using QuestionnaireResourceProvider
16496:         patterns = self.questionnaire_provider.get_patterns_by_question(question_id)
16497:         pattern_names = [p.get('pattern_id', '') for p in patterns] if patterns else []
16498:
16499:         try:
16500:             # Pydantic validation
16501:             analysis = QuestionAnalysis(
16502:                 question_id=question_id,
16503:                 question_global=question.get('question_global', 0),
16504:                 base_slot=question.get('base_slot', ''),
16505:                 scoring_modality=question.get('scoring', {}).get('modality') if isinstance(question.get('scoring'), dict) else None,
16506:                 score=result.get('score'),
16507:                 evidence=result.get('evidence', []) if isinstance(result.get('evidence'), list) else [],
16508:                 patterns_applied=pattern_names,
16509:                 recommendation=result.get('recommendation'),
16510:                 metadata={
16511:                     'dimension': question.get('dimension'),
16512:                     'policy_area': question.get('policy_area')
16513:                 }
16514:             )
16515:             analyses.append(analysis)
16516:
16517:             self.report_logger.log_validation(
16518:                 item_type="question_analysis",
16519:                 correlation_id=correlation_id,
16520:                 success=True,
16521:                 question_id=question_id
16522:             )
16523:
```

```
16524:         except Exception as e:
16525:             # Log validation failure but continue
16526:             self.report_logger.log_validation(
16527:                 item_type="question_analysis",
16528:                 correlation_id=correlation_id,
16529:                 success=False,
16530:                 error=str(e),
16531:                 question_id=question_id
16532:             )
16533:             logger.error(f"Failed to create QuestionAnalysis for {question_id}: {e}")
16534:
16535:     return analyses
16536:
16537: @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_meso_clusters")
16538: @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_meso_clusters")
16539: def _assemble_meso_clusters(
16540:     self,
16541:     execution_results: dict[str, Any]
16542: ) -> dict[str, MesoCluster]:
16543:     """Assemble meso-level cluster analyses with strict validation."""
16544:     raw_clusters = execution_results.get('meso_clusters', {})
16545:
16546:     # Handle list format from Bayesian orchestrator
16547:     if isinstance(raw_clusters, list):
16548:         # Convert list of objects to dict keyed by cluster_id
16549:         cluster_dict = {}
16550:         for item in raw_clusters:
16551:             # Handle both dicts and objects (if coming from dataclasses)
16552:             if hasattr(item, '__dict__'):
16553:                 data = item.__dict__
16554:             elif isinstance(item, dict):
16555:                 data = item
16556:             else:
16557:                 continue
16558:
16559:             c_id = data.get('cluster_id')
16560:             if c_id:
16561:                 cluster_dict[c_id] = data
16562:     raw_clusters = cluster_dict
16563:
16564:     if not isinstance(raw_clusters, dict):
16565:         logger.warning(f"meso_clusters is not a dict/list, got {type(raw_clusters).__name__}")
16566:         return {}
16567:
16568:     validated_clusters = {}
16569:     for cluster_id, data in raw_clusters.items():
16570:         try:
16571:             # Ensure data is a dict
16572:             if hasattr(data, '__dict__'):
16573:                 data = data.__dict__
16574:
16575:             if not isinstance(data, dict):
16576:                 continue
16577:
16578:             cluster = MesoCluster(
16579:                 cluster_id=str(data.get('cluster_id', cluster_id)),
```

```
16580:             raw_meso_score=float(data.get('raw_meso_score', 0.0)),
16581:             adjusted_score=float(data.get('adjusted_score', 0.0)),
16582:             dispersion_penalty=float(data.get('dispersion_penalty', 0.0)),
16583:             peer_penalty=float(data.get('peer_penalty', 0.0)),
16584:             total_penalty=float(data.get('total_penalty', 0.0)),
16585:             dispersion_metrics=data.get('dispersion_metrics', {}),
16586:             micro_scores=data.get('micro_scores', []),
16587:             metadata=data.get('metadata', {})
16588:         )
16589:         validated_clusters[cluster_id] = cluster
16590:     except Exception as e:
16591:         logger.error(f"Failed to validate meso cluster {cluster_id}: {e}")
16592:
16593:     return validated_clusters
16594:
16595: @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._assemble_macro_summary")
16596: def _assemble_macro_summary(
16597:     self,
16598:     execution_results: dict[str, Any]
16599: ) -> MacroSummary | None:
16600:     """Assemble macro-level summary with strict validation and recommendation wiring."""
16601:     raw_macro = execution_results.get('macro_summary', {})
16602:
16603:     # Handle object format
16604:     if hasattr(raw_macro, '__dict__'):
16605:         raw_macro = raw_macro.__dict__
16606:
16607:     if not isinstance(raw_macro, dict) or not raw_macro:
16608:         logger.warning("macro_summary is missing or invalid")
16609:     return None
16610:
16611:     try:
16612:         # Parse recommendations
16613:         raw_recs = raw_macro.get('recommendations', [])
16614:         validated_recs = []
16615:         for rec in raw_recs:
16616:             if isinstance(rec, str):
16617:                 validated_recs.append(Recommendation.from_string(rec))
16618:             elif isinstance(rec, dict):
16619:                 # Already structured?
16620:                 try:
16621:                     validated_recs.append(Recommendation(**rec))
16622:                 except:
16623:                     pass
16624:
16625:             return MacroSummary(
16626:                 overall_posterior=float(raw_macro.get('overall_posterior', 0.0)),
16627:                 adjusted_score=float(raw_macro.get('adjusted_score', 0.0)),
16628:                 coverage_penalty=float(raw_macro.get('coverage_penalty', 0.0)),
16629:                 dispersion_penalty=float(raw_macro.get('dispersion_penalty', 0.0)),
16630:                 contradiction_penalty=float(raw_macro.get('contradiction_penalty', 0.0)),
16631:                 total_penalty=float(raw_macro.get('total_penalty', 0.0)),
16632:                 contradiction_count=int(raw_macro.get('contradiction_count', 0)),
16633:                 recommendations=validated_recs,
16634:                 metadata=raw_macro.get('metadata', {})
16635:             )
```

```
16636:         except Exception as e:
16637:             logger.error(f"Failed to validate macro summary: {e}")
16638:             return None
16639:
16640:     @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler.export_report")
16641:     def export_report(
16642:         self,
16643:         report: AnalysisReport,
16644:         output_path: Path,
16645:         format: str = 'json'
16646:     ) -> None:
16647:         """
16648:             Export report to file.
16649:
16650:             Args:
16651:                 report: AnalysisReport to export
16652:                 output_path: Path to output file
16653:                 format: Output format ('json' or 'markdown')
16654:
16655:             Raises:
16656:                 ReportExportError: If export fails
16657:                 ReportValidationError: If format is unsupported
16658:
16659:             NOTE: This delegates I/O to factory for architectural compliance.
16660:         """
16661:         import time
16662:         start_time = time.time()
16663:         correlation_id = report.metadata.correlation_id
16664:
16665:         try:
16666:             # Delegate to factory for I/O
16667:             from farfan_pipeline.analysis.factory import save_json, write_text_file
16668:
16669:             if format == 'json':
16670:                 save_json(report.to_dict(), str(output_path))
16671:             elif format == 'markdown':
16672:                 markdown = self._format_as_markdown(report)
16673:                 write_text_file(markdown, str(output_path))
16674:             else:
16675:                 raise ReportValidationError(
16676:                     f"Unsupported format: {format}",
16677:                     details={'format': format, 'supported': ['json', 'markdown']},
16678:                     stage="export"
16679:                 )
16680:
16681:             latency_ms = (time.time() - start_time) * 1000
16682:
16683:             self.report_logger.log_operation(
16684:                 operation="export_report",
16685:                 correlation_id=correlation_id,
16686:                 success=True,
16687:                 latency_ms=latency_ms,
16688:                 output_path=str(output_path),
16689:                 format=format
16690:             )
16691:
```

```
16692:         logger.info(f"Report exported to {output_path} in {format} format")
16693:
16694:     except ReportValidationError:
16695:         raise
16696:     except Exception as e:
16697:         raise ReportExportError(
16698:             f"Failed to export report: {str(e)}",
16699:             details={'output_path': str(output_path), 'format': format, 'error': str(e)},
16700:             stage="export",
16701:             recoverable=True
16702:         ) from e
16703:
16704:     @calibrated_method("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown")
16705:     def _format_as_markdown(self, report: AnalysisReport) -> str:
16706:         """Format report as Markdown with externalized parameters."""
16707:         # Externalized parameters
16708:         # Load from calibration system if available
16709:         if _PARAM_LOADER:
16710:             preview_count = _PARAM_LOADER.get("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown").get("preview_question_count", 10)
16711:             hash_preview_length = _PARAM_LOADER.get("farfan_core.analysis.report_assembly.ReportAssembler._format_as_markdown").get("hash_preview_length", 16)
16712:
16713:         else:
16714:             preview_count = 10
16715:             hash_preview_length = 16
16716:
16717:         lines = [
16718:             f"# Policy Analysis Report: {report.metadata.plan_name}\n",
16719:             f"**Report ID:** {report.metadata.report_id}\n",
16720:             f"**Generated:** {report.metadata.generated_at}\n",
16721:             f"**Monolith Version:** {report.metadata.monolith_version}\n",
16722:             f"**Monolith Hash:** {report.metadata.monolith_hash[:hash_preview_length]}...\n",
16723:             f"**Questions Analyzed:** {report.metadata.questions_analyzed}/{report.metadata.total_questions}\n",
16724:         ]
16725:
16726:         if report.report_digest:
16727:             lines.append(f"**Report Digest:** {report.report_digest[:hash_preview_length]}...\n")
16728:
16729:         lines.append("\n## Micro-Level Analyses\n")
16730:
16731:         for analysis in report.micro_analyses[:preview_count]:
16732:             lines.append(f"\n## {analysis.question_id}\n")
16733:             lines.append(f"- **Slot:** {analysis.base_slot}\n")
16734:             lines.append(f"- **Score:** {analysis.score}\n")
16735:             lines.append(f"- **Patterns:** {', '.join(analysis.patterns_applied)}\n")
16736:
16737:         if len(report.micro_analyses) > preview_count:
16738:             lines.append(f"\n...and {len(report.micro_analyses) - preview_count} more questions_\n")
16739:
16740:         lines.append("\n## Meso-Level Clusters\n")
16741:         for cid, cluster in report.meso_clusters.items():
16742:             lines.append(f"\n## Cluster {cid}\n")
16743:             lines.append(f"- **Score:** {cluster.adjusted_score:.4f} (Raw: {cluster.raw_meso_score:.4f})\n")
16744:             lines.append(f"- **Penalties:** Total {cluster.total_penalty:.4f} (Dispersion: {cluster.dispersion_penalty:.4f}, Peer: {cluster.peer_penalty:.4f})\n")
16745:
16746:         if report.macro_summary:
```

```
16746:         lines.append("\n## Macro Summary\n")
16747:         lines.append(f"-- Overall Score:** {report.macro_summary.adjusted_score:.4f}\n")
16748:         lines.append(f"-- Contradictions:** {report.macro_summary.contradiction_count}\n")
16749:
16750:         lines.append("\n### Recommendations\n")
16751:         for rec in report.macro_summary.recommendations:
16752:             icon = "\u2728" if "CRITICAL" in rec.severity else "\u2727" if "HIGH" in rec.severity else "\u2726"
16753:             lines.append(f"-- {icon} **{rec.type}** ({rec.severity}): {rec.description}\n")
16754:     else:
16755:         lines.append("\n## Macro Summary\n")
16756:         lines.append("_No macro summary available_\n")
16757:
16758:     if report.evidence_chain_hash:
16759:         lines.append(f"\n**Evidence Chain Hash:** {report.evidence_chain_hash[:hash_preview_length]}...\n")
16760:
16761:     return "\n".join(lines)
16762:
16763:     def _compute_signal_usage_summary(
16764:         self,
16765:         execution_results: dict[str, Any],
16766:         enriched_packs: dict[str, Any]
16767:     ) -> dict[str, Any]:
16768:         """
16769:             Compute signal usage summary for report provenance (JOBFRONT 9).
16770:
16771:             Args:
16772:                 execution_results: Results from orchestrator execution
16773:                 enriched_packs: Dictionary of EnrichedSignalPack by policy_area_id
16774:
16775:             Returns:
16776:                 Signal usage summary with patterns, completeness, validation failures
16777:         """
16778:         micro_results = execution_results.get("micro_results", {})
16779:
16780:         total_patterns_available = sum(len(pack.patterns) for pack in enriched_packs.values())
16781:         total_patterns_used = 0
16782:         by_policy_area = {}
16783:         completeness_scores = []
16784:         validation_failures = []
16785:
16786:         for question_id, result in micro_results.items():
16787:             policy_area = result.get("policy_area_id")
16788:             if not policy_area or policy_area not in enriched_packs:
16789:                 continue
16790:
16791:             patterns_used = result.get("patterns_used", [])
16792:             completeness = result.get("completeness", 1.0)
16793:             validation = result.get("validation", {})
16794:
16795:             total_patterns_used += len(patterns_used)
16796:             completeness_scores.append(completeness)
16797:
16798:             # Track validation failures
16799:             if validation.get("status") == "failed" or validation.get("contract_failed"):
16800:                 validation_failures.append({
16801:                     "question_id": question_id,
```

```

16802:             "policy_area": policy_area,
16803:             "error_code": validation.get("errors", [{}])[0].get("error_code") if validation.get("errors") else None,
16804:             "remediation": validation.get("errors", [{}])[0].get("remediation") if validation.get("errors") else None
16805:         })
16806:
16807:     # Aggregate by policy area
16808:     if policy_area not in by_policy_area:
16809:         by_policy_area[policy_area] = {
16810:             "patterns_available": len(enriched_packs[policy_area].patterns),
16811:             "patterns_used": 0,
16812:             "questions_analyzed": 0,
16813:             "avg_completeness": 0.0
16814:         }
16815:
16816:     by_policy_area[policy_area]["patterns_used"] += len(patterns_used)
16817:     by_policy_area[policy_area]["questions_analyzed"] += 1
16818:
16819:     # Compute averages
16820:     for pa_id, summary in by_policy_area.items():
16821:         pa_results = [r for r in micro_results.values() if r.get("policy_area_id") == pa_id]
16822:         completeness_values = [r.get("completeness", 1.0) for r in pa_results]
16823:         summary["avg_completeness"] = sum(completeness_values) / len(completeness_values) if completeness_values else 0.0
16824:
16825:     return {
16826:         "total_patterns_available": total_patterns_available,
16827:         "total_patterns_used": total_patterns_used,
16828:         "by_policy_area": by_policy_area,
16829:         "avg_completeness": sum(completeness_scores) / len(completeness_scores) if completeness_scores else 0.0,
16830:         "validation_failures": validation_failures
16831:     }
16832:
16833:
16834: # =====
16835: # FACTORY FUNCTIONS
16836: # =====
16837:
16838: def create_report_assembler(
16839:     questionnaire_provider,
16840:     evidence_registry=None,
16841:     qmcm_recorder=None,
16842:     orchestrator=None
16843: ) -> ReportAssembler:
16844:
16845:     """Factory function to create ReportAssembler with dependencies.
16846:
16847:     Args:
16848:         questionnaire_provider: QuestionnaireResourceProvider instance
16849:         evidence_registry: Optional EvidenceRegistry
16850:         qmcm_recorder: Optional QMCMRecorder
16851:         orchestrator: Optional Orchestrator
16852:
16853:     Returns:
16854:         Configured ReportAssembler
16855:
16856:     Raises:
16857:         ReportValidationError: If required dependencies are missing

```

```
16858: """
16859:     return ReportAssembler(
16860:         questionnaire_provider=questionnaire_provider,
16861:         evidence_registry=evidence_registry,
16862:         qmcm_recorder=qmcm_recorder,
16863:         orchestrator=orchestrator
16864:     )
16865:
16866:
16867: # =====
16868: # MODULE EXPORTS
16869: # =====
16870:
16871: __all__ = [
16872:     # Exceptions
16873:     'ReportAssemblyException',
16874:     'ReportValidationError',
16875:     'ReportIntegrityError',
16876:     'ReportExportError',
16877:     # Contracts
16878:     'ReportMetadata',
16879:     'QuestionAnalysis',
16880:     'AnalysisReport',
16881:     # Main Classes
16882:     'ReportAssembler',
16883:     'ReportLogger',
16884:     # Factory Functions
16885:     'create_report_assembler',
16886:     # Utilities
16887:     'compute_content_digest',
16888:     'utc_now_iso',
16889: ]
16890:
16891:
16892: # =====
16893: # IN-SCRIPT VALIDATION
16894: # =====
16895:
16896: if __name__ == "__main__":
16897:     print("=" * 70)
16898:     print("Report Assembly Module - Validation Suite")
16899:     print("=" * 70)
16900:
16901:     # Test 1: Domain-specific exceptions
16902:     print("\n1. Testing domain-specific exceptions:")
16903:     try:
16904:         raise ReportValidationError(
16905:             "Test validation error",
16906:             details={'field': 'test'},
16907:             stage="validation"
16908:         )
16909:     except ReportValidationError as e:
16910:         print(f"  \u234\u223 ReportValidationError: {e.event_id[:8]}... - {e.message}")
16911:         print(f"  \u234\u223 Structured dict: {list(e.to_dict().keys())}")
16912:
16913:     # Test 2: Pydantic contract validation
```

```
16914:     print("\n2. Testing Pydantic contract validation:")
16915:     try:
16916:         # Invalid hash (not 64 chars)
16917:         ReportMetadata(
16918:             report_id="test-001",
16919:             monolith_version="1.0",
16920:             monolith_hash="invalid",
16921:             plan_name="Test Plan",
16922:             total_questions=10,
16923:             questions_analyzed=5
16924:         )
16925:         print("    \u234\u227 Expected validation error for invalid hash")
16926:     except Exception as e:
16927:         print(f"    \u234\u223 Caught validation error: {type(e).__name__}")
16928:
16929:     # Valid metadata
16930:     valid_hash = "a" * 64
16931:     metadata = ReportMetadata(
16932:         report_id="test-001",
16933:         monolith_version="1.0",
16934:         monolith_hash=valid_hash,
16935:         plan_name="Test Plan",
16936:         total_questions=10,
16937:         questions_analyzed=5
16938:     )
16939:     print(f"    \u234\u223 Valid metadata created: {metadata.report_id}")
16940:
16941:     # Test 3: Cryptographic digest
16942:     print("\n3. Testing cryptographic digest:")
16943:     test_content = {"key": "value", "number": 42}
16944:     digest = compute_content_digest(test_content)
16945:     print(f"    \u234\u223 Digest computed: {digest[:16]}... (length: {len(digest)} )")
16946:     assert len(digest) == 64, "Digest must be 64 characters"
16947:     print("    \u234\u223 Digest length validated")
16948:
16949:     # Test 4: Report digest verification
16950:     print("\n4. Testing report digest verification:")
16951:     micro_analysis = QuestionAnalysis(
16952:         question_id="Q001",
16953:         question_global=1,
16954:         base_slot="slot1",
16955:         score=0.85
16956:     )
16957:
16958:     meso_cluster = MesoCluster(
16959:         cluster_id="CL01",
16960:         raw_meso_score=0.8,
16961:         adjusted_score=0.75,
16962:         dispersion_penalty=0.05,
16963:         peer_penalty=0.0,
16964:         total_penalty=0.05
16965:     )
16966:
16967:     macro_summary = MacroSummary(
16968:         overall_posterior=0.75,
16969:         adjusted_score=0.7,
```

```
16970:     coverage_penalty=0.05,
16971:     dispersion_penalty=0.0,
16972:     contradiction_penalty=0.0,
16973:     total_penalty=0.05,
16974:     contradiction_count=0,
16975:     recommendations=[  
16976:         Recommendation(type="RISK", severity="LOW", description="Monitor closely")  
16977:     ]  
16978: )  
16979:  
16980: report = AnalysisReport(  
16981:     metadata=metadata,  
16982:     micro_analyses=[micro_analysis],  
16983:     meso_clusters={"CL01": meso_cluster},  
16984:     macro_summary=macro_summary  
16985: )  
16986:  
16987: report_digest = report.compute_digest()  
16988: print(f"  \u2192 Report digest: {report_digest[:16]}...")  
16989:  
16990: # Create report with digest  
16991: report_with_digest = AnalysisReport(  
16992:     metadata=metadata,  
16993:     micro_analyses=[micro_analysis],  
16994:     meso_clusters={"CL01": meso_cluster},  
16995:     macro_summary=macro_summary,  
16996:     report_digest=report_digest  
16997: )  
16998:  
16999: is_valid = report_with_digest.verify_digest()  
17000: print(f"  \u2192 Digest verification: {is_valid}")  
17001:  
17002: # Test 5: Structured logging  
17003: print("\n5. Testing structured logging:")  
17004: test_logger = ReportLogger("test")  
17005: test_logger.log_operation(  
17006:     operation="test_operation",  
17007:     correlation_id=metadata.correlation_id,  
17008:     success=True,  
17009:     latency_ms=12.345,  
17010:     custom_field="test_value"  
17011: )  
17012: print("  \u2192 Structured log emitted")  
17013:  
17014: print("\n" + "=" * 70)  
17015: print("All validation tests passed!")  
17016: print("=" * 70)  
17017:  
17018:  
17019:  
17020: ======  
17021: FILE: src/farfan_pipeline/analysis/retry_handler.py  
17022: ======  
17023:  
17024: from enum import Enum  
17025: import time
```

```
17026: import logging
17027: from functools import wraps
17028:
17029: class DependencyType(Enum):
17030:     SPACY_MODEL = "spaCy_model"
17031:     PDF_PARSER = "pdf_parser"
17032:
17033: def get_retry_handler():
17034:     return RetryHandler()
17035:
17036: class RetryHandler:
17037:     def __init__(self, max_retries=3, delay=1):
17038:         self.max_retries = max_retries
17039:         self.delay = delay
17040:         self.logger = logging.getLogger(self.__class__.__name__)
17041:
17042:     def with_retry(self, dependency_type, operation_name, exceptions):
17043:         def decorator(func):
17044:             @wraps(func)
17045:             def wrapper(*args, **kwargs):
17046:                 retries = 0
17047:                 while retries < self.max_retries:
17048:                     try:
17049:                         return func(*args, **kwargs)
17050:                     except exceptions as e:
17051:                         retries += 1
17052:                         self.logger.warning(
17053:                             f"Operation '{operation_name}' for dependency '{dependency_type.value}' failed. "
17054:                             f"Retrying ({retries})/{self.max_retries})... Error: {e}"
17055:                         )
17056:                         time.sleep(self.delay)
17057:                         self.logger.error(
17058:                             f"Operation '{operation_name}' for dependency '{dependency_type.value}' "
17059:                             f"failed after {self.max_retries} retries."
17060:                         )
17061:                         raise
17062:                 return wrapper
17063:             return decorator
17064:
17065:
17066:
17067: =====
17068: FILE: src/farfan_pipeline/analysis/scoring/__init__.py
17069: =====
17070:
17071: """
17072: Scoring Package
17073:
17074: Implements TYPE_A through TYPE_F scoring modalities with strict validation
17075: and reproducible results.
17076:
17077: NOTE: Evidence and MicroQuestionScorer are NOT in this package.
17078: They exist in the parent MODULE: farfan_core/analysis/scoring.py
17079: Import them directly from there: 'from farfan_core.analysis.scoring import Evidence'
17080: """
17081:
```

```
17082: # Import from this package's scoring.py
17083: from farfan_pipeline.analysis.scoring import (
17084:     EvidenceStructureError,
17085:     ModalityConfig,
17086:     ModalityValidationException,
17087:     QualityLevel,
17088:     ScoredResult,
17089:     ScoringError,
17090:     ScoringModality,
17091:     ScoringValidator,
17092:     apply_scoring,
17093:     determine_quality_level,
17094: )
17095:
17096: __all__ = [
17097:     "EvidenceStructureError",
17098:     "ModalityConfig",
17099:     "ModalityValidationException",
17100:     "QualityLevel",
17101:     "ScoredResult",
17102:     "ScoringError",
17103:     "ScoringModality",
17104:     "ScoringValidator",
17105:     "apply_scoring",
17106:     "determine_quality_level",
17107: ]
17108:
17109: # ARCHITECTURAL NOTE FOR MAINTAINERS:
17110: # Evidence and MicroQuestionScorer live in farfan_core/analysis/scoring.py (module)
17111: # This __init__.py is for farfan_core/analysis/scoring/ (package)
17112: # These are SEPARATE namespaces. Import Evidence from the module directly.
17113:
17114:
17115:
17116: =====
17117: FILE: src/farfant_pipeline/analysis/scoring/scoring.py
17118: =====
17119:
17120: """
17121: Scoring Module - TYPE_A through TYPE_F Modality Implementation
17122:
17123: This module implements the scoring system for the SAAAAAA policy analysis framework.
17124: It provides:
17125: - Application of 6 scoring modalities (TYPE_A through TYPE_F)
17126: - Validation of evidence structure vs modality
17127: - Assignment of quality levels
17128: - Structured logging with strict abortability
17129: - Reproducible ScoredResult outputs
17130:
17131: Preconditions:
17132: - Evidence and modality must be declared
17133: - Evidence structure must match modality requirements
17134:
17135: Invariants:
17136: - Score range is maintained per modality definition
17137: - Evidence structure is validated before scoring
```

```
17138:  
17139: Postconditions:  
17140: - ScoredResult is reproducible with same inputs  
17141: - No fallback or partial heuristic scoring  
17142: """  
17143:  
17144: from __future__ import annotations  
17145:  
17146: import hashlib  
17147: import json  
17148: import logging  
17149: import math  
17150: from dataclasses import asdict, dataclass, field  
17151: from datetime import datetime, timezone  
17152: from decimal import ROUND_DOWN, ROUND_HALF_EVEN, ROUND_HALF_UP, Decimal, InvalidOperation  
17153: from enum import Enum  
17154: from numbers import Real  
17155: from typing import Any, ClassVar  
17156: from farfan_pipeline.core.parameters import ParameterLoaderV2  
17157: from farfan_pipeline.core.calibration.decorators import calibrated_method  
17158:  
17159: logger = logging.getLogger(__name__)  
17160:  
17161: class ScoringModality(Enum):  
17162:     """Scoring modality types."""  
17163:     TYPE_A = "TYPE_A" # Bayesian: Numerical claims, gaps, risks  
17164:     TYPE_B = "TYPE_B" # DAG: Causal chains, ToC completeness  
17165:     TYPE_C = "TYPE_C" # Coherence: Inverted contradictions  
17166:     TYPE_D = "TYPE_D" # Pattern: Baseline data, formalization  
17167:     TYPE_E = "TYPE_E" # Financial: Budget traceability  
17168:     TYPE_F = "TYPE_F" # Beach: Mechanism inference, plausibility  
17169:  
17170: class QualityLevel(Enum):  
17171:     """Quality level classifications."""  
17172:     EXCELENTE = "EXCELENTE"  
17173:     BUENO = "BUENO"  
17174:     ACCEPTABLE = "ACCEPTABLE"  
17175:     INSUFICIENTE = "INSUFICIENTE"  
17176:  
17177: class ScoringError(Exception):  
17178:     """Base exception for scoring errors."""  
17179:     pass  
17180:  
17181: class ModalityValidationError(ScoringError):  
17182:     """Exception raised when evidence structure doesn't match modality requirements."""  
17183:     pass  
17184:  
17185: class EvidenceStructureError(ScoringError):  
17186:     """Exception raised when evidence structure is invalid."""  
17187:     pass  
17188:  
17189: @dataclass(frozen=True)  
17190: class ScoredResult:  
17191:     """  
17192:     Reproducible scored result for a question.  
17193:
```

```
17194:     Attributes:
17195:         question_global: Global question number (1-300)
17196:         base_slot: Question slot identifier
17197:         policy_area: Policy area ID (PA01-PA10)
17198:         dimension: Dimension ID (DIM01-DIM06)
17199:         modality: Scoring modality used (TYPE_A through TYPE_F)
17200:         score: Raw score value
17201:         normalized_score: Normalized score (0-1)
17202:         quality_level: Quality level classification
17203:         evidence_hash: SHA-256 hash of evidence for reproducibility
17204:         metadata: Additional scoring metadata
17205:         timestamp: ISO timestamp of scoring
17206:         """
17207:         question_global: int
17208:         base_slot: str
17209:         policy_area: str
17210:         dimension: str
17211:         modality: str
17212:         score: float
17213:         normalized_score: float
17214:         quality_level: str
17215:         evidence_hash: str
17216:         metadata: dict[str, Any] = field(default_factory=dict)
17217:         timestamp: str = field(default_factory=lambda: datetime.now(timezone.utc).isoformat().replace("+00:00", "Z"))
17218:
17219:     @calibrated_method("farfan_core.analysis.scoring.ScoringResult.to_dict")
17220:     def to_dict(self) -> dict[str, Any]:
17221:         """Convert to dictionary representation."""
17222:         return asdict(self)
17223:
17224:     @staticmethod
17225:     def compute_evidence_hash(evidence: dict[str, Any]) -> str:
17226:         """
17227:             Compute reproducible hash of evidence.
17228:
17229:             Args:
17230:                 evidence: Evidence dictionary
17231:
17232:             Returns:
17233:                 SHA-256 hash as hex string
17234:         """
17235:         canonical = json.dumps(evidence, ensure_ascii=False, sort_keys=True, separators=(", ", ":"), indent=4)
17236:         return hashlib.sha256(canonical.encode("utf-8")).hexdigest()
17237:
17238:     @dataclass
17239:     class ModalityConfig:
17240:         """
17241:             Configuration for a scoring modality.
17242:
17243:             Attributes:
17244:                 name: Modality name
17245:                 description: Modality description
17246:                 score_range: Min and max score values
17247:                 rounding_mode: Rounding mode (half_up, bankers, truncate)
17248:                 rounding_precision: Decimal precision for rounding
17249:                 required_evidence_keys: Required keys in evidence
```

```
17250:     expected_elements: Expected number of elements (if applicable)
17251:     deterministic: Whether scoring is deterministic
17252:     """
17253:     name: str
17254:     description: str
17255:     score_range: tuple[float, float]
17256:     rounding_mode: str = "half_up"
17257:     rounding_precision: int = 2
17258:     required_evidence_keys: list[str] = field(default_factory=list)
17259:     expected_elements: int | None = None
17260:     deterministic: bool = True
17261:
17262:     @calibrated_method("farfan_core.analysis.scoring.ModalityConfig.validate_evidence")
17263:     def validate_evidence(self, evidence: dict[str, Any]) -> None:
17264:         """
17265:             Validate evidence structure against modality requirements.
17266:
17267:             Args:
17268:                 evidence: Evidence dictionary to validate
17269:
17270:             Raises:
17271:                 EvidenceStructureError: If evidence is missing required keys
17272:                 ModalityValidationError: If evidence structure doesn't match modality
17273:         """
17274:         if not isinstance(evidence, dict):
17275:             raise EvidenceStructureError(
17276:                 f"Evidence must be a dictionary, got {type(evidence).__name__}"
17277:             )
17278:
17279:         # Check required keys
17280:         missing_keys = [key for key in self.required_evidence_keys if key not in evidence]
17281:         if missing_keys:
17282:             raise EvidenceStructureError(
17283:                 f"Evidence missing required keys for {self.name}: {missing_keys}"
17284:             )
17285:
17286:         # Validate expected elements if applicable
17287:         if self.expected_elements is not None:
17288:             elements = evidence.get("elements", [])
17289:             if not isinstance(elements, list):
17290:                 raise ModalityValidationError(
17291:                     f"{self.name} requires 'elements' to be a list, got {type(elements).__name__}"
17292:                 )
17293:
17294: class ScoringValidator:
17295:     """Validates evidence structure against modality requirements."""
17296:
17297:     # Modality configurations
17298:     MODALITY_CONFIGS: ClassVar[dict[ScoringModality, ModalityConfig]] = {
17299:         ScoringModality.TYPE_A: ModalityConfig(
17300:             name="TYPE_A",
17301:             description="Bayesian: Numerical claims, gaps, risks",
17302:             score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L182_25", 0.0), 3.0),
17303:             required_evidence_keys=["elements", "confidence"],
17304:             expected_elements=4,
17305:         ),
```

```
17306:     ScoringModality.TYPE_B: ModalityConfig(
17307:         name="TYPE_B",
17308:         description="DAG: Causal chains, ToC completeness",
17309:         score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L189_25", 0.0), 3.0),
17310:         required_evidence_keys=["elements", "completeness"],
17311:         expected_elements=3,
17312:     ),
17313:     ScoringModality.TYPE_C: ModalityConfig(
17314:         name="TYPE_C",
17315:         description="Coherence: Inverted contradictions",
17316:         score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L196_25", 0.0), 3.0),
17317:         required_evidence_keys=["elements", "coherence_score"],
17318:         expected_elements=2,
17319:     ),
17320:     ScoringModality.TYPE_D: ModalityConfig(
17321:         name="TYPE_D",
17322:         description="Pattern: Baseline data, formalization",
17323:         score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L203_25", 0.0), 3.0),
17324:         required_evidence_keys=["elements", "pattern_matches"],
17325:         expected_elements=3,
17326:     ),
17327:     ScoringModality.TYPE_E: ModalityConfig(
17328:         name="TYPE_E",
17329:         description="Financial: Budget traceability",
17330:         score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L210_25", 0.0), 3.0),
17331:         required_evidence_keys=["elements", "traceability"],
17332:     ),
17333:     ScoringModality.TYPE_F: ModalityConfig(
17334:         name="TYPE_F",
17335:         description="Beach: Mechanism inference, plausibility",
17336:         score_range=(ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L216_25", 0.0), 3.0),
17337:         required_evidence_keys=["elements", "plausibility"],
17338:     ),
17339: }
17340:
17341: @classmethod
17342: def validate(
17343:     cls,
17344:     evidence: dict[str, Any],
17345:     modality: ScoringModality,
17346: ) -> None:
17347:     """
17348:     Validate evidence structure against modality.
17349:
17350:     Args:
17351:         evidence: Evidence dictionary
17352:         modality: Scoring modality
17353:
17354:     Raises:
17355:         ModalityValidationError: If validation fails
17356:
17357:     Note:
17358:         This function has strict abortability - any validation failure
17359:             will raise an exception and halt processing.
17360:     """
17361:     config = cls.MODALITY_CONFIGS.get(modality)
```

```
17362:     if not config:
17363:         raise ModalityValidationError(f"Unknown modality: {modality}")
17364:
17365:     logger.info(f"Validating evidence for {modality.value}")
17366:
17367:     try:
17368:         config.validate_evidence(evidence)
17369:         logger.info(f"\u234\u223 Evidence validation passed for {modality.value}")
17370:     except (EvidenceStructureError, ModalityValidationError) as e:
17371:         logger.exception(f"\u234\u227 Evidence validation failed for {modality.value}: {e}")
17372:         raise
17373:
17374:     @classmethod
17375:     def get_config(cls, modality: ScoringModality) -> ModalityConfig:
17376:         """Get configuration for a modality."""
17377:         config = cls.MODALITY_CONFIGS.get(modality)
17378:         if not config:
17379:             raise ModalityValidationError(f"Unknown modality: {modality}")
17380:         return config
17381:
17382:     def clamp(value: float, lower: float, upper: float) -> float:
17383:         """Clamp *value* to the inclusive range '[lower, upper]'."""
17384:
17385:         if lower > upper:
17386:             raise ValueError("Lower bound cannot exceed upper bound")
17387:
17388:         return min(max(value, lower), upper)
17389:
17390:     def apply_rounding(
17391:         value: float,
17392:         mode: str = "half_up",
17393:         precision: int = 2,
17394:     ) -> float:
17395:         """
17396:             Apply rounding to a numeric value.
17397:
17398:             Args:
17399:                 value: Value to round
17400:                 mode: Rounding mode (half_up, bankers, truncate)
17401:                 precision: Decimal precision
17402:
17403:             Returns:
17404:                 Rounded value
17405:         """
17406:         if precision < 0:
17407:             raise ValueError("Precision must be non-negative")
17408:
17409:         decimal_value = Decimal(str(value))
17410:         quantize_exp = Decimal(10) ** -precision
17411:
17412:         if mode == "half_up":
17413:             rounding_mode = ROUND_HALF_UP
17414:         elif mode == "bankers":
17415:             rounding_mode = ROUND_HALF_EVEN
17416:         elif mode == "truncate":
17417:             rounding_mode = ROUND_DOWN
```

```
17418:     else:
17419:         raise ValueError(f"Unknown rounding mode: {mode}")
17420:
17421:     try:
17422:         rounded = decimal_value.quantize(quantize_exp, rounding=rounding_mode)
17423:     except InvalidOperation as exc:
17424:         raise ValueError(f"Failed to round value {value}: {exc}") from exc
17425:
17426:     return float(rounded)
17427:
17428: def _validate_quality_thresholds(thresholds: dict[str, float]) -> dict[str, float]:
17429:     """Validate custom quality thresholds.
17430:
17431:     Returns a copy of *thresholds* with float values if validation succeeds.
17432:     """
17433:
17434:     if not isinstance(thresholds, dict):
17435:         raise ValueError("Quality thresholds must be provided as a dictionary")
17436:
17437:     required_keys = ("EXCELENTE", "BUENO", "ACEPTABLE")
17438:     missing = [key for key in required_keys if key not in thresholds]
17439:     if missing:
17440:         raise ValueError(f"Missing quality thresholds for: {', '.join(missing)}")
17441:
17442:     validated: dict[str, float] = {}
17443:     for key in required_keys:
17444:         value = thresholds[key]
17445:
17446:         if isinstance(value, bool) or not isinstance(value, (int, float, Decimal, Real)):
17447:             raise ValueError(
17448:                 f"Threshold for {key} must be a real number between 0 and 1"
17449:             )
17450:
17451:         numeric_value = float(value)
17452:         if math.isnan(numeric_value) or math.isinf(numeric_value):
17453:             raise ValueError(f"Threshold for {key} cannot be NaN or infinite")
17454:
17455:         if not ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L335_15", 0.0) <= numeric_value <=
17456:             ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L335_39", 1.0):
17457:                 raise ValueError(
17458:                     f"Threshold for {key} must be between 0 and 1 inclusive"
17459:                 )
17460:
17461:         validated[key] = numeric_value
17462:
17463:     if not (
17464:         validated["EXCELENTE"] >= validated["BUENO"] >= validated["ACEPTABLE"]
17465:     ):
17466:         raise ValueError(
17467:             "Quality thresholds must satisfy EXCELENTE >= BUENO >= ACEPTABLE"
17468:         )
17469:
17470:     return validated
17471: def score_type_a(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
17472:     """
```

```
17473:     Score TYPE_A evidence: Bayesian numerical claims, gaps, risks.
17474:
17475:     Expects:
17476:         - elements: List of up to 4 elements
17477:         - confidence: Bayesian confidence score (0-1)
17478:
17479:     Scoring:
17480:         - Count elements (max 4)
17481:         - Weight by confidence
17482:         - Scale to 0-3 range
17483:
17484:     Args:
17485:         evidence: Evidence dictionary
17486:         config: Modality configuration
17487:
17488:     Returns:
17489:         Tuple of (score, metadata)
17490:     """
17491:     elements = evidence.get("elements", [])
17492:     confidence = evidence.get("confidence", ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L372_44", 0.0))
17493:
17494:     if not isinstance(elements, list):
17495:         raise ModalityValidationError("TYPE_A: 'elements' must be a list")
17496:
17497:     if not isinstance(confidence, (int, float)) or not (0 <= confidence <= 1):
17498:         raise ModalityValidationError("TYPE_A: 'confidence' must be a number between 0 and 1")
17499:
17500:     # Count valid elements (up to expected)
17501:     element_count = min(len(elements), config.expected_elements or 4)
17502:
17503:     max_elements = config.expected_elements or 4
17504:     max_score = config.score_range[1] if config.score_range else 3.0
17505:     min_score = config.score_range[0] if config.score_range else ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L385_65", 0.0)
17506:
17507:     # Calculate raw score: count weighted by confidence, scale to range
17508:     max_elements = config.expected_elements if config.expected_elements is not None else 4
17509:     scale = config.score_range[1] if config.score_range else 3.0
17510:     raw_score = (element_count / max(1, max_elements)) * scale * confidence
17511:
17512:     # Clamp to valid range
17513:     score = max(min_score, min(max_score, raw_score))
17514:
17515:     metadata = {
17516:         "element_count": element_count,
17517:         "confidence": confidence,
17518:         "raw_score": raw_score,
17519:         "expected_elements": config.expected_elements,
17520:         "max_score": max_score,
17521:     }
17522:
17523:     logger.info(
17524:         f"TYPE_A score: {score:.2f} "
17525:         f"(elements={element_count}, confidence={confidence:.2f})"
17526:     )
```

```
17527:  
17528:     return score, metadata  
17529:  
17530: def score_type_b(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:  
17531:     """  
17532:         Score TYPE_B evidence: DAG causal chains, ToC completeness.  
17533:  
17534:     Expects:  
17535:         - elements: List of causal chain elements (up to 3)  
17536:         - completeness: DAG completeness score (0-1)  
17537:  
17538:     Scoring:  
17539:         - Count causal elements (max 3)  
17540:         - Weight by completeness  
17541:         - Each element worth 1 point  
17542:  
17543:     Args:  
17544:         evidence: Evidence dictionary  
17545:         config: Modality configuration  
17546:  
17547:     Returns:  
17548:         Tuple of (score, metadata)  
17549:     """  
17550:     elements = evidence.get("elements", [])  
17551:     completeness = evidence.get("completeness", ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L  
431_48", 0.0))  
17552:  
17553:     if not isinstance(elements, list):  
17554:         raise ModalityValidationError("TYPE_B: 'elements' must be a list")  
17555:  
17556:     if not isinstance(completeness, (int, float)) or not (0 <= completeness <= 1):  
17557:         raise ModalityValidationError("TYPE_B: 'completeness' must be a number between 0 and 1")  
17558:  
17559:     # Count valid elements (up to expected)  
17560:     element_count = min(len(elements), config.expected_elements or 3)  
17561:  
17562:     # Calculate raw score: each element worth 1 point, weighted by completeness  
17563:     raw_score = float(element_count) * completeness  
17564:  
17565:     # Clamp to valid range  
17566:     score = max(config.score_range[0], min(config.score_range[1], raw_score))  
17567:  
17568:     metadata = {  
17569:         "element_count": element_count,  
17570:         "completeness": completeness,  
17571:         "raw_score": raw_score,  
17572:         "expected_elements": config.expected_elements,  
17573:     }  
17574:  
17575:     logger.info(  
17576:         f"TYPE_B score: {score:.2f} "  
17577:         f"(elements={element_count}, completeness={completeness:.2f})"  
17578:     )  
17579:  
17580:     return score, metadata  
17581:
```

```
17582: def score_type_c(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
17583:     """
17584:         Score TYPE_C evidence: Coherence via inverted contradictions.
17585:
17586:         Expects:
17587:             - elements: List of coherence elements (up to 2)
17588:             - coherence_score: Inverted contradiction score (0-1, higher is better)
17589:
17590:         Scoring:
17591:             - Count coherence elements (max 2)
17592:             - Scale by coherence score
17593:             - Scale to 0-3 range
17594:
17595:         Args:
17596:             evidence: Evidence dictionary
17597:             config: Modality configuration
17598:
17599:         Returns:
17600:             Tuple of (score, metadata)
17601:             """
17602:     elements = evidence.get("elements", [])
17603:     coherence_score = evidence.get("coherence_score", ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L483_54", 0.0))
17604:
17605:     if not isinstance(elements, list):
17606:         raise ModalityValidationError("TYPE_C: 'elements' must be a list")
17607:
17608:     if not isinstance(coherence_score, (int, float)) or not (0 <= coherence_score <= 1):
17609:         raise ModalityValidationError("TYPE_C: 'coherence_score' must be a number between 0 and 1")
17610:
17611:     # Count valid elements (up to expected)
17612:     element_count = min(len(elements), config.expected_elements or 2)
17613:
17614:     # Calculate raw score: scale elements to range, weighted by coherence
17615:     raw_score = (element_count / 2.0) * 3.0 * coherence_score
17616:
17617:     # Clamp to valid range
17618:     score = max(config.score_range[0], min(config.score_range[1], raw_score))
17619:
17620:     metadata = {
17621:         "element_count": element_count,
17622:         "coherence_score": coherence_score,
17623:         "raw_score": raw_score,
17624:         "expected_elements": config.expected_elements,
17625:     }
17626:
17627:     logger.info(
17628:         f"TYPE_C score: {score:.2f} "
17629:         f"(elements={element_count}, coherence={coherence_score:.2f})"
17630:     )
17631:
17632:     return score, metadata
17633:
17634: def score_type_d(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
17635:     """
17636:         Score TYPE_D evidence: Pattern matching for baseline data.
```

```
17637:  
17638:    Expects:  
17639:        - elements: List of pattern matches (up to 3)  
17640:        - pattern_matches: Number of successful pattern matches  
17641:  
17642:    Scoring:  
17643:        - Count pattern matches (max 3)  
17644:        - Weight by match quality if available  
17645:        - Scale to 0-3 range  
17646:  
17647:    Args:  
17648:        evidence: Evidence dictionary  
17649:        config: Modality configuration  
17650:  
17651:    Returns:  
17652:        Tuple of (score, metadata)  
17653:    """  
17654:    elements = evidence.get("elements", [])  
17655:    pattern_matches = evidence.get("pattern_matches", 0)  
17656:  
17657:    if not isinstance(elements, list):  
17658:        raise ModalityValidationError("TYPE_D: 'elements' must be a list")  
17659:  
17660:    if not isinstance(pattern_matches, (int, float)) or pattern_matches < 0:  
17661:        raise ModalityValidationError("TYPE_D: 'pattern_matches' must be a non-negative number")  
17662:  
17663:    # Count valid elements (up to expected)  
17664:    element_count = min(len(elements), config.expected_elements or 3)  
17665:  
17666:    # Use actual pattern matches if available, otherwise use element count  
17667:    match_count = min(pattern_matches, element_count) if pattern_matches > 0 else element_count  
17668:  
17669:    # Calculate raw score: scale to 0-3 range  
17670:    raw_score = (match_count / 3.0) * 3.0  
17671:  
17672:    # Clamp to valid range  
17673:    score = max(config.score_range[0], min(config.score_range[1], raw_score))  
17674:  
17675:    metadata = {  
17676:        "element_count": element_count,  
17677:        "pattern_matches": match_count,  
17678:        "raw_score": raw_score,  
17679:        "expected_elements": config.expected_elements,  
17680:    }  
17681:  
17682:    logger.info(  
17683:        f"TYPE_D score: {score:.2f} "  
17684:        f"(elements={element_count}, matches={match_count})"  
17685:    )  
17686:  
17687:    return score, metadata  
17688:  
17689: def score_type_e(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:  
17690:    """  
17691:        Score TYPE_E evidence: Financial budget traceability.  
17692:
```

```
17693:     Expects:
17694:         - elements: List of budget elements
17695:         - traceability: Boolean or numeric traceability score
17696:
17697:     Scoring:
17698:         - Boolean presence check
17699:         - If numeric traceability provided, use that
17700:             - Scale to 0-3 range
17701:
17702:     Args:
17703:         evidence: Evidence dictionary
17704:         config: Modality configuration
17705:
17706:     Returns:
17707:         Tuple of (score, metadata)
17708: """
17709:     elements = evidence.get("elements", [])
17710:     traceability = evidence.get("traceability", False)
17711:
17712:     if not isinstance(elements, list):
17713:         raise ModalityValidationError("TYPE_E: 'elements' must be a list")
17714:
17715:     # Handle both boolean and numeric traceability
17716:     if isinstance(traceability, bool):
17717:         traceability_score = ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L597_29", 1.0) if tr
aceability else ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L597_54", 0.0)
17718:     elif isinstance(traceability, (int, float)):
17719:         if not (0 <= traceability <= 1):
17720:             raise ModalityValidationError("TYPE_E: numeric 'traceability' must be between 0 and 1")
17721:         traceability_score = float(traceability)
17722:     else:
17723:         raise ModalityValidationError("TYPE_E: 'traceability' must be boolean or numeric")
17724:
17725:     # Count valid elements
17726:     element_count = len(elements)
17727:     has_elements = element_count > 0
17728:
17729:     # Calculate raw score: presence check weighted by traceability
17730:     raw_score = 3.0 * traceability_score if has_elements else ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence",
"auto_param_L610_62", 0.0)
17731:
17732:     # Clamp to valid range
17733:     score = max(config.score_range[0], min(config.score_range[1], raw_score))
17734:
17735:     metadata = {
17736:         "element_count": element_count,
17737:         "traceability": traceability_score,
17738:         "raw_score": raw_score,
17739:         "has_elements": has_elements,
17740:     }
17741:
17742:     logger.info(
17743:         f"TYPE_E score: {score:.2f} "
17744:         f"(elements={element_count}, traceability={traceability_score:.2f})"
17745:     )
17746:
```

```
17747:     return score, metadata
17748:
17749: def score_type_f(evidence: dict[str, Any], config: ModalityConfig) -> tuple[float, dict[str, Any]]:
17750:     """
17751:         Score TYPE_F evidence: Beach mechanism inference and plausibility.
17752:
17753:         Expects:
17754:             - elements: List of mechanism elements
17755:             - plausibility: Plausibility score (0-1)
17756:
17757:         Scoring:
17758:             - Continuous scale based on plausibility
17759:             - Weight by element presence
17760:             - Scale to 0-3 range
17761:
17762:         Args:
17763:             evidence: Evidence dictionary
17764:             config: Modality configuration
17765:
17766:         Returns:
17767:             Tuple of (score, metadata)
17768:             """
17769:         elements = evidence.get("elements", [])
17770:         plausibility = evidence.get("plausibility", ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L650_48", 0.0))
17771:
17772:         if not isinstance(elements, list):
17773:             raise ModalityValidationError("TYPE_F: 'elements' must be a list")
17774:
17775:         if not isinstance(plausibility, (int, float)) or not (0 <= plausibility <= 1):
17776:             raise ModalityValidationError("TYPE_F: 'plausibility' must be a number between 0 and 1")
17777:
17778:         # Count valid elements
17779:         element_count = len(elements)
17780:
17781:         # Calculate raw score: continuous scale weighted by plausibility
17782:         raw_score = 3.0 * plausibility if element_count > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L662_61", 0.0)
17783:
17784:         # Clamp to valid range
17785:         score = max(config.score_range[0], min(config.score_range[1], raw_score))
17786:
17787:         metadata = {
17788:             "element_count": element_count,
17789:             "plausibility": plausibility,
17790:             "raw_score": raw_score,
17791:         }
17792:
17793:         logger.info(
17794:             f"TYPE_F score: {score:.2f} "
17795:             f"(elements={element_count}, plausibility={plausibility:.2f})"
17796:         )
17797:
17798:     return score, metadata
17799:
17800: # Scoring function registry
```

```
17801: SCORING_FUNCTIONS = {
17802:     ScoringModality.TYPE_A: score_type_a,
17803:     ScoringModality.TYPE_B: score_type_b,
17804:     ScoringModality.TYPE_C: score_type_c,
17805:     ScoringModality.TYPE_D: score_type_d,
17806:     ScoringModality.TYPE_E: score_type_e,
17807:     ScoringModality.TYPE_F: score_type_f,
17808: }
17809:
17810: def determine_quality_level(
17811:     normalized_score: float,
17812:     thresholds: dict[str, float] | None = None,
17813: ) -> QualityLevel:
17814:     """
17815:     Determine quality level from normalized score.
17816:
17817:     Args:
17818:         normalized_score: Score normalized to 0-1 range
17819:         thresholds: Optional custom thresholds
17820:
17821:     Returns:
17822:         Quality level
17823:
17824:     Note:
17825:         Default thresholds:
17826:         - EXCELENTE: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L706_24", 0.85)
17827:         - BUENO: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L707_20", 0.70)
17828:         - ACEPTABLE: >= ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L708_24", 0.55)
17829:         - INSUFICIENTE: < ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L709_26", 0.55)
17830:     """
17831:     if thresholds is None:
17832:         thresholds = {
17833:             "EXCELENTE": ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L713_25", 0.85),
17834:             "BUENO": ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L714_21", 0.70),
17835:             "ACEPTABLE": ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L715_25", 0.55),
17836:         }
17837:
17838:     thresholds = _validate_quality_thresholds(thresholds)
17839:
17840:     # Clamp score to account for minor floating-point drift
17841:     normalized_score = clamp(float(normalized_score), ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L721_54", 0.0), ParameterLoaderV2.get("farfan_core.analysis.scoring.ModalityConfig.validate_evidence", "auto_param_L721_59", 1.0))
17842:
17843:     if normalized_score >= thresholds["EXCELENTE"]:
17844:         return QualityLevel.EXCELENTE
17845:     elif normalized_score >= thresholds["BUENO"]:
17846:         return QualityLevel.BUENO
17847:     elif normalized_score >= thresholds["ACEPTABLE"]:
17848:         return QualityLevel.ACEPTABLE
17849:     else:
17850:         return QualityLevel.INSUFICIENTE
17851:
17852: def apply_scoring(
17853:     question_global: int,
17854:     base_slot: str,
17855:     policy_area: str,
```

```
17856:     dimension: str,
17857:     evidence: dict[str, Any],
17858:     modality: str,
17859:     quality_thresholds: dict[str, float] | None = None,
17860: ) -> ScoredResult:
17861: """
17862:     Apply scoring to evidence using specified modality.
17863:
17864:     This is the main entry point for scoring. It:
17865:     1. Validates evidence structure against modality
17866:     2. Applies modality-specific scoring function
17867:     3. Normalizes score to 0-1 range
17868:     4. Determines quality level
17869:     5. Returns reproducible ScoredResult
17870:
17871: Args:
17872:     question_global: Global question number (1-300)
17873:     base_slot: Question slot identifier
17874:     policy_area: Policy area ID (PA01-PA10)
17875:     dimension: Dimension ID (DIM01-DIM06)
17876:     evidence: Evidence dictionary
17877:     modality: Scoring modality (TYPE_A through TYPE_F)
17878:     quality_thresholds: Optional custom quality thresholds
17879:
17880: Returns:
17881:     ScoredResult
17882:
17883: Raises:
17884:     ModalityValidationError: If evidence validation fails
17885:     ScoringError: If scoring fails
17886:
17887: Note:
17888:     This function has strict abortability. Any validation or scoring
17889:     error will raise an exception and halt processing. No fallback
17890:     or partial scoring is performed.
17891: """
17892: logger.info(
17893:     f"Scoring question {question_global} ({base_slot}) "
17894:     f"using {modality}"
17895: )
17896:
17897: # Parse modality
17898: try:
17899:     modality_enum = ScoringModality(modality)
17900: except ValueError as e:
17901:     raise ModalityValidationError(
17902:         f"Invalid modality: {modality}. "
17903:         f"Must be one of: {[m.value for m in ScoringModality]}"
17904:     ) from e
17905:
17906: # Validate evidence structure
17907: ScoringValidator.validate(evidence, modality_enum)
17908:
17909: # Get modality configuration
17910: config = ScoringValidator.get_config(modality_enum)
17911:
```

```
17912:     # Get scoring function
17913:     scoring_func = SCORING_FUNCTIONS.get(modality_enum)
17914:     if not scoring_func:
17915:         raise ScoringError(f"No scoring function for {modality}")
17916:
17917:     # Apply scoring
17918:     try:
17919:         score, metadata = scoring_func(evidence, config)
17920:     except (ModalityValidationError, EvidenceStructureError, ScoringError) as e:
17921:         logger.exception(f"Scoring failed for {modality}: {e}")
17922:         raise ScoringError(f"Scoring failed for {modality}: {e}") from e
17923:     except Exception as e:
17924:         logger.exception(f"Unexpected error in scoring {modality}: {e}")
17925:         raise ScoringError(f"Unexpected error in scoring {modality}: {e}") from e
17926:
17927:     # Apply rounding
17928:     rounded_score = apply_rounding(
17929:         score,
17930:         mode=config.rounding_mode,
17931:         precision=config.rounding_precision,
17932:     )
17933:
17934:     min_score, max_score = config.score_range
17935:     if max_score <= min_score:
17936:         raise ScoringError(
17937:             f"Invalid score range for {modality}: {config.score_range}"
17938:         )
17939:
17940:     # Guard against errant modality implementations
17941:     clamped_score = clamp(rounded_score, min_score, max_score)
17942:     score_clamped = not math.isclose(clamped_score, rounded_score, rel_tol=1e-9, abs_tol=1e-9)
17943:
17944:     # Normalize score to 0-1 range
17945:     normalized_score = (clamped_score - min_score) / (max_score - min_score)
17946:     normalized_score = clamp(normalized_score, ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L826_47", 0.0), ParameterLoaderV2.get("farfan_core.analysis.scoring.scoring.ModalityConfig.validate_evidence", "auto_param_L826_52", 1.0))
17947:
17948:     # Determine quality level
17949:     quality_level = determine_quality_level(normalized_score, quality_thresholds)
17950:
17951:     # Compute evidence hash for reproducibility
17952:     evidence_hash = ScoredResult.compute_evidence_hash(evidence)
17953:
17954:     # Build result
17955:     result = ScoredResult(
17956:         question_global=question_global,
17957:         base_slot=base_slot,
17958:         policy_area=policy_area,
17959:         dimension=dimension,
17960:         modality=modality,
17961:         score=rounded_score,
17962:         normalized_score=normalized_score,
17963:         quality_level=quality_level.value,
17964:         evidence_hash=evidence_hash,
17965:         metadata={
17966:             **metadata,
```

```
17967:         "score_range": config.score_range,
17968:         "rounding_mode": config.rounding_mode,
17969:         "rounding_precision": config.rounding_precision,
17970:         "score_clamped": score_clamped,
17971:     },
17972: )
17973:
17974:     logger.info(
17975:         f"\u234\u223 Scoring complete: score={rounded_score:.2f}, "
17976:         f"normalized={normalized_score:.2f}, quality={quality_level.value}"
17977:     )
17978:
17979:     return result
17980:
17981: __all__ = [
17982:     "ScoringModality",
17983:     "QualityLevel",
17984:     "ScoringError",
17985:     "ModalityValidationError",
17986:     "EvidenceStructureError",
17987:     "ScoredResult",
17988:     "ModalityConfig",
17989:     "ScoringValidator",
17990:     "apply_scoring",
17991:     "determine_quality_level",
17992: ]
17993:
17994:
17995:
17996: =====
17997: FILE: src/farfan_pipeline/analysis/scoring.py
17998: =====
17999:
18000: """
18001: SCORING MODULE - Question Scoring According to Questionnaire Monolith
18002: =====
18003: File: scoring.py
18004: Code: SC
18005: Purpose: Apply scoring modalities to question results
18006:
18007: This module implements the scoring system for policy assessment questions.
18008: All scoring modalities and quality thresholds are defined in the questionnaire
18009: monolith specification (lines 34512-34607).
18010:
18011: SCORING MODALITIES (6 types):
18012: -----
18013: 1. TYPE_A: Count 4 elements and scale to 0-3 (threshold=0.7 ratio)
18014:     - Used when 4 specific policy elements must be present
18015:     - Threshold: 70% of elements must be found to receive partial credit
18016:
18017: 2. TYPE_B: Count up to 3 elements, each worth 1 point
18018:     - Used for independent policy components
18019:     - Each element contributes equally to the final score
18020:
18021: 3. TYPE_C: Count 2 elements and scale to 0-3 (threshold=0.5 ratio)
18022:     - Used when 2 critical policy elements must be present
```

```
18023:     - Threshold: 50% of elements must be found to receive partial credit
18024:
18025: 4. TYPE_D: Count 3 elements, weighted [0.4, 0.3, 0.3]
18026:     - Used when policy elements have different importance
18027:     - First element has highest weight (40%), others equal (30% each)
18028:
18029: 5. TYPE_E: Boolean presence check
18030:     - Binary scoring: element is present (3 points) or absent (0 points)
18031:
18032: 6. TYPE_F: Semantic matching with cosine similarity (normalized_continuous)
18033:     - Uses text similarity to assess policy alignment
18034:     - Continuous score based on semantic similarity (0.0-1.0 range)
18035:
18036: QUALITY LEVELS:
18037: -----
18038: Quality levels are determined from normalized scores (0.0-1.0 scale):
18039: - EXCELLENT: ≥ 0.85 (85th percentile) - green indicator
18040: - GOOD: ≥ 0.70 (70th percentile) - blue indicator
18041: - ACCEPTABLE: ≥ 0.55 (55th percentile) - yellow indicator
18042: - INSUFFICIENT: < 0.55 (below 55th percentile) - red indicator
18043:
18044: CORE METHODS:
18045: -----
18046: 1. MicroQuestionScorer.score_type_a() - TYPE_A scoring logic
18047: 2. MicroQuestionScorer.score_type_b() - TYPE_B scoring logic
18048: 3. MicroQuestionScorer.score_type_c() - TYPE_C scoring logic
18049: 4. MicroQuestionScorer.score_type_d() - TYPE_D scoring logic
18050: 5. MicroQuestionScorer.score_type_e() - TYPE_E scoring logic
18051: 6. MicroQuestionScorer.score_type_f() - TYPE_F scoring logic
18052: 7. MicroQuestionScorer.apply_scoring_modality() - Dispatcher for modalities
18053: 8. MicroQuestionScorer.determine_quality_level() - Maps scores to quality levels
18054:
18055: DATA FLOW:
18056: -----
18057: Input: QuestionResult with evidence from Phase 2 evaluation
18058: Output: ScoredResult with score (0-3 range) and quality level classification
18059:
18060: REFERENCE:
18061: -----
18062: Questionnaire monolith specification lines 34512-34607
18063: """
18064:
18065: import logging
18066: from dataclasses import dataclass, field
18067: from enum import Enum
18068: from typing import Any
18069:
18070: import numpy as np
18071: from farfan_pipeline.core.parameters import ParameterLoaderV2
18072: from farfan_pipeline.core.calibration.decorators import calibrated_method
18073:
18074: logger = logging.getLogger(__name__)
18075:
18076: # =====
18077: # ENUMS - EXACTOS DEL MONOLITH
18078: # =====
```

```
18079:  
18080: class ScoringModality(Enum):  
18081:     """Modalidades de scoring del monolith (lÃ±ea 34535)."""  
18082:     TYPE_A = "TYPE_A" # Count 4 elements and scale to 0-3  
18083:     TYPE_B = "TYPE_B" # Count up to 3 elements, each worth 1 point  
18084:     TYPE_C = "TYPE_C" # Count 2 elements and scale to 0-3  
18085:     TYPE_D = "TYPE_D" # Count 3 elements, weighted  
18086:     TYPE_E = "TYPE_E" # Boolean presence check  
18087:     TYPE_F = "TYPE_F" # Semantic matching with cosine similarity  
18088:  
18089: class QualityLevel(Enum):  
18090:     """Niveles de calidad micro (lÃ±ea 34513)."""  
18091:     EXCELENTE = "EXCELENTE" # >= 0.85  
18092:     BUENO = "BUENO" # >= 0.70  
18093:     ACEPTABLE = "ACEPTABLE" # >= 0.55  
18094:     INSUFICIENTE = "INSUFICIENTE" # < 0.55  
18095:  
18096: # =====  
18097: # DATACLASSES  
18098: # =====  
18099:  
18100: @dataclass  
18101: class ScoringConfig:  
18102:     """  
18103:         Scoring configuration extracted from questionnaire monolith specification.  
18104:  
18105:         This configuration defines all parameters for the six scoring modalities  
18106:         and quality level thresholds. All values are derived from the questionnaire  
18107:         monolith specification (lines 34512-34607).  
18108:  
18109:     Attributes:  
18110:         TYPE_A Configuration (line 34568):  
18111:             type_a_threshold: Ratio threshold for partial credit (0.0-1.0 scale, default: 0.7)  
18112:                 Elements found / expected must exceed this to receive credit  
18113:             type_a_max_score: Maximum score achievable (default: 3.0 points)  
18114:             type_a_expected_elements: Number of elements expected (default: 4 elements)  
18115:  
18116:         TYPE_B Configuration (line 34574):  
18117:             type_b_max_score: Maximum score achievable (default: 3.0 points)  
18118:             type_b_max_elements: Maximum elements to count (default: 3 elements)  
18119:  
18120:         TYPE_C Configuration (line 34580):  
18121:             type_c_threshold: Ratio threshold for partial credit (0.0-1.0 scale, default: 0.5)  
18122:                 Elements found / expected must exceed this to receive credit  
18123:             type_c_max_score: Maximum score achievable (default: 3.0 points)  
18124:             type_c_expected_elements: Number of elements expected (default: 2 elements)  
18125:  
18126:         TYPE_D Configuration (line 34586):  
18127:             type_d_weights: Importance weights for each element (0.0-1.0 scale per weight,  
18128:                         must sum to 1.0, default: [0.4, 0.3, 0.3])  
18129:                 First element weighted 40%, second and third 30% each  
18130:             type_d_max_score: Maximum score achievable (default: 3.0 points)  
18131:             type_d_expected_elements: Number of elements expected (default: 3 elements)  
18132:  
18133:         TYPE_E Configuration (line 34596):  
18134:             type_e_max_score: Maximum score achievable (default: 3.0 points)
```

```
18135:             Binary: full score if present, 0 if absent
18136:
18137:     TYPE_F Configuration (line 34601):
18138:         type_f_max_score: Maximum score achievable (default: 3.0 points)
18139:         type_f_normalization: Normalization method for similarity scores (default: "minmax")
18140:             Options: "minmax", "zscore", "none"
18141:
18142:     Quality Level Thresholds (line 34513):
18143:         level_excelente_min: Minimum normalized score for EXCELLENT (0.0-1.0 scale, default: 0.85)
18144:         level_bueno_min: Minimum normalized score for GOOD (0.0-1.0 scale, default: 0.70)
18145:         level_aceptable_min: Minimum normalized score for ACCEPTABLE (0.0-1.0 scale, default: 0.55)
18146:         level_insuficiente_min: Minimum normalized score for INSUFFICIENT (0.0-1.0 scale, default: 0.0)
18147: """
18148:
18149: # TYPE_A config (line 34568)
18150: type_a_threshold: float = 0.7 # Ratio (0.0-1.0): proportion of elements required
18151: type_a_max_score: float = 3.0 # Points: maximum achievable score
18152: type_a_expected_elements: int = 4 # Count: number of policy elements to check
18153:
18154: # TYPE_B config (line 34574)
18155: type_b_max_score: float = 3.0 # Points: maximum achievable score
18156: type_b_max_elements: int = 3 # Count: maximum elements to score
18157:
18158: # TYPE_C config (line 34580)
18159: type_c_threshold: float = 0.5 # Ratio (0.0-1.0): proportion of elements required
18160: type_c_max_score: float = 3.0 # Points: maximum achievable score
18161: type_c_expected_elements: int = 2 # Count: number of policy elements to check
18162:
18163: # TYPE_D config (line 34586)
18164: type_d_weights: list[float] = field(default_factory=lambda: [0.4, 0.3, 0.3]) # Weights (sum to 1.0): element importance
18165: type_d_max_score: float = 3.0 # Points: maximum achievable score
18166: type_d_expected_elements: int = 3 # Count: number of policy elements to check
18167:
18168: # TYPE_E config (line 34596)
18169: type_e_max_score: float = 3.0 # Points: maximum achievable score (binary: 3.0 or 0.0)
18170:
18171: # TYPE_F config (line 34601)
18172: type_f_max_score: float = 3.0 # Points: maximum achievable score
18173: type_f_normalization: str = "minmax" # Method: "minmax", "zscore", or "none"
18174:
18175: # Quality levels (line 34513) - All thresholds are normalized scores (0.0-1.0 scale)
18176: level_excelente_min: float = 0.85 # Ratio (0.0-1.0): minimum for EXCELLENT quality
18177: level_bueno_min: float = 0.70 # Ratio (0.0-1.0): minimum for GOOD quality
18178: level_aceptable_min: float = 0.55 # Ratio (0.0-1.0): minimum for ACCEPTABLE quality
18179: level_insuficiente_min: float = 0.0 # Ratio (0.0-1.0): minimum for INSUFFICIENT quality
18180:
18181: @dataclass
18182: class Evidence:
18183: """
18184:     Evidencia extraÃ±a para una pregunta.
18185:     Producida por evaluadores en FASE 2.
18186: """
18187:     elements_found: list[str] = field(default_factory=list)
18188:     confidence_scores: list[float] = field(default_factory=list)
18189:     semantic_similarity: float | None = None
18190:     pattern_matches: dict[str, int] = field(default_factory=dict)
```

```
18191:     metadata: dict[str, Any] = field(default_factory=dict)
18192:
18193: @dataclass
18194: class ScoredResult:
18195:     """
18196:         Resultado con score aplicado.
18197:         Output de este módulo.
18198:     """
18199:     question_id: str
18200:     question_global: int
18201:     scoring_modality: ScoringModality
18202:     raw_score: float # 0-3
18203:     normalized_score: float # 0-1 (raw_score / 3.0)
18204:     quality_level: QualityLevel
18205:     quality_color: str # "green", "blue", "yellow", "red"
18206:     evidence: Evidence
18207:     scoring_details: dict[str, Any] = field(default_factory=dict)
18208:
18209: # =====
18210: # CLASE: MicroQuestionScorer
18211: # =====
18212:
18213: class MicroQuestionScorer:
18214:     """
18215:         Aplicador de modalidades de scoring segúrn monolith.
18216:
18217:         Responsabilidades:
18218:             - Aplicar TYPE_A, TYPE_B, TYPE_C, TYPE_D, TYPE_E, TYPE_F
18219:             - Calcular score 0-3
18220:             - Determinar nivel de calidad (EXCELENTE/BUENO/ACEPTABLE/INSUFICIENTE)
18221:     """
18222:
18223:     def __init__(self, config: ScoringConfig | None = None) -> None:
18224:         """
18225:             Inicializa scorer con configuracióñ del monolith.
18226:
18227:             Args:
18228:                 config: Configuracióñ de scoring (defaults del monolith si None)
18229:             """
18230:             self.config = config or ScoringConfig()
18231:             self.logger = logger
18232:
18233: # =====
18234: # MÁ\211TODO 1: SCORE TYPE_A
18235: # =====
18236:
18237: @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a")
18238: def score_type_a(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18239:     """
18240:         MÁ\211TODO 1: TYPE_A - Count 4 elements and scale to 0-3.
18241:
18242:         ESPECIFICACIÓ223N (lÁ-nea 34568 del monolith):
18243:             - Aggregation: "presence_threshold"
18244:             - Threshold: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L245_21", 0.7)
18245:             - Max_score: 3
18246:             - Expected_elements: 4
```

```
18247:  
18248:     LÃ223GICA:  
18249:         1. Contar elementos encontrados (expected: 4)  
18250:         2. Calcular ratio = found / 4  
18251:         3. Si ratio >= ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L252_23", 0.7): aplicar escala pro  
porcional  
18252:         4. Si ratio < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L253_22", 0.7): penalizar fuertemen  
te  
18253:  
18254:     ESCALA:  
18255:         - 4/4 elementos (100%) \206\222 3.0  
18256:         - 3/4 elementos (75%) \206\222 2.25  
18257:         - 2/4 elementos (50%) \206\222 penalizado  
18258:         - 1/4 elementos (25%) \206\222 penalizado  
18259:         - 0/4 elementos (0%) \206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L260_31", 0.0)  
18260:  
18261:     Args:  
18262:         evidence: Evidencia extraÃda con elements_found  
18263:  
18264:     Returns:  
18265:         Tuple de (score, details)  
18266:         """  
18267:     elements_found = len(evidence.elements_found)  
18268:     expected = self.config.type_a_expected_elements  
18269:     threshold = self.config.type_a_threshold  
18270:     max_score = self.config.type_a_max_score  
18271:  
18272:         # Calcular ratio  
18273:         ratio = elements_found / expected if expected > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_  
param_L274_63", 0.0)  
18274:  
18275:         # Aplicar threshold del monolith  
18276:         if ratio >= threshold:  
18277:             # Escala proporcional: ratio * max_score  
18278:             score = ratio * max_score  
18279:         else:  
18280:             # PenalizaciÃ³n: escala cuadrÃ¡tica para ratios bajos  
18281:             score = (ratio / threshold) * (ratio * max_score)  
18282:  
18283:         # Clip al rango [0, max_score]  
18284:         score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_a", "auto_param_L285_20", 0.0), min(max_score, score)  
)  
18285:  
18286:     details = {  
18287:         'modality': 'TYPE_A',  
18288:         'elements_found': elements_found,  
18289:         'expected_elements': expected,  
18290:         'ratio': ratio,  
18291:         'threshold': threshold,  
18292:         'threshold_met': ratio >= threshold,  
18293:         'raw_score': score,  
18294:         'formula': 'ratio * max_score if ratio >= threshold else penalized'  
18295:     }  
18296:  
18297:     self.logger.debug(f"TYPE_A: {elements_found}/{expected} elementos ({ratio:.2f}) \206\222 score={score:.2f}")  
18298:
```

```
18299:         return score, details
18300:
18301:     # =====
18302:     # MÃ\211TODO 2: SCORE_TYPE_B
18303:     # =====
18304:
18305:     @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b")
18306:     def score_type_b(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18307:         """
18308:             MÃ\211TODO 2: TYPE_B - Count up to 3 elements, each worth 1 point.
18309:
18310:             ESPECIFICACIÃ\223N (lÃ-nea 34574 del monolith):
18311:             - Aggregation: "binary_sum"
18312:             - Max_score: 3
18313:             - Max_elements: 3
18314:
18315:             LÃ\223GICA:
18316:             1. Contar elementos encontrados (max: 3)
18317:             2. Cada elemento = 1 punto
18318:             3. Score = min(elements_found, 3)
18319:
18320:             ESCALA:
18321:             - 3+ elementos Ã\206\222 3.0
18322:             - 2 elementos Ã\206\222 2.0
18323:             - 1 elemento Ã\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b", "auto_param_L324_23", 1.0)
18324:             - 0 elementos Ã\206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_b", "auto_param_L325_24", 0.0)
18325:
18326:             Args:
18327:                 evidence: Evidencia extraÃ-da con elements_found
18328:
18329:             Returns:
18330:                 Tuple de (score, details)
18331:             """
18332:             elements_found = len(evidence.elements_found)
18333:             max_elements = self.config.type_b_max_elements
18334:             max_score = self.config.type_b_max_score
18335:
18336:             # Binary sum: cada elemento vale 1 punto, hasta max_elements
18337:             score = min(float(elements_found), max_elements)
18338:
18339:             # Asegurar que no excede max_score
18340:             score = min(score, max_score)
18341:
18342:             details = {
18343:                 'modality': 'TYPE_B',
18344:                 'elements_found': elements_found,
18345:                 'max_elements': max_elements,
18346:                 'raw_score': score,
18347:                 'formula': 'min(elements_found, 3)'
18348:             }
18349:
18350:             self.logger.debug(f"TYPE_B: {elements_found} elementos Ã\206\222 score={score:.2f}")
18351:
18352:             return score, details
18353:
18354:     # =====
```

```
18355:     # MÃ©TODO 3: SCORE TYPE_C
18356:     # =====
18357:
18358:     @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c")
18359:     def score_type_c(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18360:         """
18361:             MÃ©TODO 3: TYPE_C - Count 2 elements and scale to 0-3.
18362:
18363:             ESPECIFICACIÃ©N (lÃ±nea 34580 del monolith):
18364:             - Aggregation: "presence_threshold"
18365:             - Threshold: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L366_21", 0.5)
18366:             - Max_score: 3
18367:             - Expected_elements: 2
18368:
18369:             LÃ©GICA:
18370:             1. Contar elementos encontrados (expected: 2)
18371:             2. Calcular ratio = found / 2
18372:             3. Si ratio >= ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L373_23", 0.5): aplicar escala proporcional
18373:             4. Si ratio < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L374_22", 0.5): penalizar
18374:
18375:             ESCALA:
18376:             - 2/2 elementos (100%) → 3.0
18377:             - 1/2 elementos (50%) → 1.5
18378:             - 0/2 elementos (0%) → 0.0 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L379_31", 0.0)
18379:
18380:             Args:
18381:                 evidence: Evidencia extraÃ±a con elements_found
18382:
18383:             Returns:
18384:                 Tuple de (score, details)
18385:             """
18386:             elements_found = len(evidence.elements_found)
18387:             expected = self.config.type_c_expected_elements
18388:             threshold = self.config.type_c_threshold
18389:             max_score = self.config.type_c_max_score
18390:
18391:             # Calcular ratio
18392:             ratio = elements_found / expected if expected > 0 else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L393_63", 0.0)
18393:
18394:             # Aplicar threshold del monolith
18395:             if ratio >= threshold:
18396:                 # Escala proporcional
18397:                 score = ratio * max_score
18398:             else:
18399:                 # PenalizaciÃ³n cuadrÃ¡tica
18400:                 score = (ratio / threshold) * (ratio * max_score)
18401:
18402:             # Clip al rango [0, max_score]
18403:             score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_c", "auto_param_L404_20", 0.0), min(max_score, score))
18404:
18405:             details = {
18406:                 'modality': 'TYPE_C',
18407:                 'elements_found': elements_found,
```

```

18408:         'expected_elements': expected,
18409:         'ratio': ratio,
18410:         'threshold': threshold,
18411:         'threshold_met': ratio >= threshold,
18412:         'raw_score': score,
18413:         'formula': 'ratio * max_score if ratio >= threshold else penalized'
18414:     }
18415:
18416:     self.logger.debug(f"TYPE_C: {elements_found}/{expected} elementos ({ratio:.2f}) \u206\222 score={score:.2f}")
18417:
18418:     return score, details
18419:
18420: # =====
18421: # M\u00e1\211TODO 4: SCORE TYPE_D
18422: # =====
18423:
18424: @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d")
18425: def score_type_d(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18426:
18427:     """\nM\u00e1\211TODO 4: TYPE_D - Count 3 elements, weighted [ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_55", 0.4), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_60", 0.3), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L428_65", 0.3)].\n
18428:
18429:     ESPECIFICACI\u2022N (l\u00e1-nea 34586 del monolith):
18430:     - Aggregation: "weighted_sum"
18431:     - Weights: [ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_20", 0.4), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_25", 0.3), ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L432_30", 0.3)]
18432:     - Max_score: 3
18433:     - Expected_elements: 3
18434:
18435: L\u00e1\223GICA:
18436: 1. Se esperan 3 elementos con importancia diferente
18437: 2. Elemento 1: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L438_28", 0.4) (m\u00e1s importante)
18438: 3. Elemento 2: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L439_28", 0.3)
18439: 4. Elemento 3: peso ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L440_28", 0.3)
18440: 5. Score = (sum of weights for found elements) * max_score
18441:
18442: ESCALA:
18443: - 3 elementos (todos) \u206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L444_44", 1.0) \u206\222 3.0
18444: - 2 elementos (ej: elem1+elem2) \u206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L445_54", 0.7) \u206\222 2.1
18445: - 1 elemento (ej: elem1) \u206\222 weights_sum=ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L446_47", 0.4) \u206\222 1.2
18446: - 0 elementos \u206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L447_24", 0.0)
18447:
18448: Args:
18449:     evidence: Evidencia extra\u00e1da con elements_found y confidence_scores
18450:
18451: Returns:
18452:     Tuple de (score, details)
18453: """
18454: elements_found = len(evidence.elements_found)
18455: expected = self.config.type_d_expected_elements

```

```

18456:     weights = self.config.type_d_weights
18457:     max_score = self.config.type_d_max_score
18458:
18459:     # Calcular suma ponderada
18460:     # Asumimos que elements_found estĂ; ordenado por importancia
18461:     # o usamos confidence_scores si estĂ;n disponibles
18462:     if evidence.confidence_scores and len(evidence.confidence_scores) >= elements_found:
18463:         # Ordenar por confidence (descendente) y aplicar pesos
18464:         sorted_confidences = sorted(evidence.confidence_scores[:elements_found], reverse=True)
18465:         weighted_sum = sum(
18466:             conf * weights[i]
18467:             for i, conf in enumerate(sorted_confidences)
18468:             if i < len(weights)
18469:         )
18470:     else:
18471:         # Sin confidence scores: asumir presencia binaria
18472:         weighted_sum = sum(weights[:min(elements_found, len(weights))])
18473:
18474:     # Score = weighted_sum * max_score
18475:     score = weighted_sum * max_score
18476:
18477:     # Clip al rango [0, max_score]
18478:     score = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_d", "auto_param_L479_20", 0.0), min(max_score, score))
18479:
18480:     details = {
18481:         'modality': 'TYPE_D',
18482:         'elements_found': elements_found,
18483:         'expected_elements': expected,
18484:         'weights': weights,
18485:         'weighted_sum': weighted_sum,
18486:         'raw_score': score,
18487:         'formula': 'weighted_sum * max_score'
18488:     }
18489:
18490:     self.logger.debug(f"TYPE_D: {elements_found}/{expected} elementos, weighted_sum={weighted_sum:.2f} \u2192 score={score:.2f}")
18491:
18492:     return score, details
18493:
18494: # =====
18495: # MÃ\211TODO 5: SCORE TYPE_E
18496: # =====
18497:
18498: @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e")
18499: def score_type_e(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18500:     """
18501:     MÃ\211TODO 5: TYPE_E - Boolean presence check.
18502:
18503:     ESPECIFICACIÃ\223N (lÃ-nea 34596 del monolith):
18504:     - Aggregation: "binary_presence"
18505:     - Max_score: 3
18506:
18507:     LÃ\223GICA:
18508:     1. Verificar si existe evidencia (binario: sÃ-/no)
18509:     2. Si existe: 3.0
18510:     3. Si no existe: ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L511_25", 0.0)

```

```
18511:
18512:     ESCALA:
18513:         - Evidencia presente \206\222 3.0
18514:         - Evidencia ausente \206\222 ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L515_30", 0.0)
18515:
18516:     Args:
18517:         evidence: Evidencia extraÃ-da
18518:
18519:     Returns:
18520:         Tuple de (score, details)
18521:         """
18522:         max_score = self.config.type_e_max_score
18523:
18524:         # Verificar presencia de cualquier evidencia
18525:         has_evidence = (
18526:             len(evidence.elements_found) > 0 or
18527:             bool(evidence.pattern_matches) or
18528:             (evidence.semantic_similarity is not None and evidence.semantic_similarity > ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L529_89", 0.5))
18529:         )
18530:
18531:         # Binary: todo o nada
18532:         score = max_score if has_evidence else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L533_47", 0.0)
18533:
18534:         details = {
18535:             'modality': 'TYPE_E',
18536:             'has_evidence': has_evidence,
18537:             'elements_found': len(evidence.elements_found),
18538:             'pattern_matches': len(evidence.pattern_matches),
18539:             'semantic_similarity': evidence.semantic_similarity,
18540:             'raw_score': score,
18541:             'formula': 'max_score if has_evidence else ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_e", "auto_param_L542_55", 0.0)'
18542:         }
18543:
18544:         self.logger.debug(f"TYPE_E: evidencia={'presente' if has_evidence else 'ausente'} \206\222 score={score:.2f}")
18545:
18546:         return score, details
18547:
18548:     # =====
18549:     # MÃ\211TODO 6: SCORE TYPE_F
18550:     # =====
18551:
18552:     @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f")
18553:     def score_type_f(self, evidence: Evidence) -> tuple[float, dict[str, Any]]:
18554:         """
18555:             MÃ\211TODO 6: TYPE_F - Semantic matching with cosine similarity.
18556:
18557:             ESPECIFICACIÃ\223N (lÃ-nea 34601 del monolith):
18558:             - Aggregation: "normalized_continuous"
18559:             - Normalization: "minmax"
18560:             - Max_score: 3
18561:
18562:             LÃ\223GICA:
18563:             1. Usar semantic_similarity (rango 0-1)
```

```

18564:     2. Normalizar con minmax
18565:     3. Score = normalized_similarity * max_score
18566:
18567:     ESCALA:
18568:     - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L569_23", 1.0) à\206\222 3.0
18569:     - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L570_23", 0.75) à\206\222 2.25
18570:     - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L571_23", 0.5) à\206\222 1.5
18571:     - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L572_23", 0.25) à\206\222 Parameter
rLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L572_30", 0.75)
18572:     - Similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L573_23", 0.0) à\206\222 Parameter
LoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L573_29", 0.0)
18573:
18574:     Args:
18575:         evidence: Evidencia con semantic_similarity
18576:
18577:     Returns:
18578:         Tuple de (score, details)
18579:     """
18580:     max_score = self.config.type_f_max_score
18581:
18582:     # Obtener similarity
18583:     if evidence.semantic_similarity is not None:
18584:         similarity = evidence.semantic_similarity
18585:     # Fallback: calcular promedio de confidence_scores
18586:     elif evidence.confidence_scores:
18587:         similarity = float(np.mean(evidence.confidence_scores))
18588:     else:
18589:         similarity = ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "similarity", 0.0) # Refactored
18590:
18591:     # NormalizaciÃ³n minmax (ya estÃ¡ en rango 0-1)
18592:     normalized_similarity = max(ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L593_36", 0.0), min(P
arameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.score_type_f", "auto_param_L593_45", 1.0), similarity))
18593:
18594:     # Score continuo
18595:     score = normalized_similarity * max_score
18596:
18597:     details = {
18598:         'modality': 'TYPE_F',
18599:         'semantic_similarity': similarity,
18600:         'normalized_similarity': normalized_similarity,
18601:         'raw_score': score,
18602:         'formula': 'normalized_similarity * max_score'
18603:     }
18604:
18605:     self.logger.debug(f"TYPE_F: similarity={similarity:.3f} à\206\222 score={score:.2f}")
18606:
18607:     return score, details
18608:
18609: # -----
18610: # MÀ\211TODO 7: APPLY SCORING MODALITY (ORQUESTADOR)
18611: # -----
18612:
18613: def apply_scoring_modality(
18614:     self,
18615:     question_id: str,
18616:     question_global: int,

```

```
18617:         modality: ScoringModality,
18618:         evidence: Evidence
18619:     ) -> ScoredResult:
18620:         """
18621:         MÃ\211TODO 7: Aplica la modalidad de scoring correspondiente.
18622:
18623:         ORQUESTADOR que delega a mÃ©todos 1-6 segÃ³n modality.
18624:
18625:     Args:
18626:         question_id: ID de pregunta (ej: "Q001")
18627:         question_global: NÃºmero global (1-305)
18628:         modality: Modalidad de scoring
18629:         evidence: Evidencia extraÃ±a
18630:
18631:     Returns:
18632:         ScoredResult con score 0-3 y nivel de calidad
18633:         """
18634:         self.logger.info(f"Aplicando scoring {modality.value} a {question_id}")
18635:
18636:         # Delegar a mÃ©todo especÃ¡fico
18637:         if modality == ScoringModality.TYPE_A:
18638:             raw_score, details = self.score_type_a(evidence)
18639:
18640:         elif modality == ScoringModality.TYPE_B:
18641:             raw_score, details = self.score_type_b(evidence)
18642:
18643:         elif modality == ScoringModality.TYPE_C:
18644:             raw_score, details = self.score_type_c(evidence)
18645:
18646:         elif modality == ScoringModality.TYPE_D:
18647:             raw_score, details = self.score_type_d(evidence)
18648:
18649:         elif modality == ScoringModality.TYPE_E:
18650:             raw_score, details = self.score_type_e(evidence)
18651:
18652:         elif modality == ScoringModality.TYPE_F:
18653:             raw_score, details = self.score_type_f(evidence)
18654:
18655:     else:
18656:         raise ValueError(f"Modalidad desconocida: {modality}")
18657:
18658:         # Normalizar a 0-1
18659:         normalized_score = raw_score / 3.0
18660:
18661:         # Determinar nivel de calidad
18662:         quality_level, quality_color = self.determine_quality_level(normalized_score)
18663:
18664:         # Construir resultado
18665:         scored_result = ScoredResult(
18666:             question_id=question_id,
18667:             question_global=question_global,
18668:             scoring_modality=modality,
18669:             raw_score=raw_score,
18670:             normalized_score=normalized_score,
18671:             quality_level=quality_level,
18672:             quality_color=quality_color,
```

```
18673:         evidence=evidence,
18674:         scoring_details=details
18675:     )
18676:
18677:     self.logger.info(
18678:         f"\u234\u223 {question_id}: score={raw_score:.2f}/3.0 "
18679:         f"({normalized_score:.2%}), nivel={quality_level.value}"
18680:     )
18681:
18682:     return scored_result
18683:
18684: # =====
18685: # M\u00e1\u211TODO 8: DETERMINE QUALITY LEVEL
18686: # =====
18687:
18688: @calibrated_method("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level")
18689: def determine_quality_level(self, normalized_score: float) -> tuple[QualityLevel, str]:
18690:     """
18691:         M\u00e1\u211TODO 8: Determina nivel de calidad seg\u00f3n umbrales del monolith.
18692:
18693:         UMBRALES (l\u00e1nea 34513 del monolith):
18694:             - EXCELENTE: \u211d ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L695_23", 0.85) (verde)
18695:             - BUENO: \u211d ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L696_19", 0.70) (azul)
18696:             - ACEPTABLE: \u211d ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L697_23", 0.55) (amarillo)
18697:             - INSUFICIENTE: < ParameterLoaderV2.get("farfan_core.analysis.scoring.MicroQuestionScorer.determine_quality_level", "auto_param_L698_26", 0.55) (rojo)
18698:
18699:         Args:
18700:             normalized_score: Score en rango 0-1
18701:
18702:         Returns:
18703:             Tuple de (QualityLevel, color)
18704:             """
18705:             if normalized_score >= self.config.level_excelente_min:
18706:                 return QualityLevel.EXCELENTE, "green"
18707:
18708:             elif normalized_score >= self.config.level_bueno_min:
18709:                 return QualityLevel.BUENO, "blue"
18710:
18711:             elif normalized_score >= self.config.level_aceptable_min:
18712:                 return QualityLevel.ACCEPTABLE, "yellow"
18713:
18714:             else:
18715:                 return QualityLevel.INSUFICIENTE, "red"
18716:
18717: # =====
18718: # FUNCION \u223N DE CONVENIENCIA
18719: # =====
18720:
18721: def score_question(
18722:     question_id: str,
18723:     question_global: int,
18724:     modality_str: str,
18725:     evidence_dict: dict[str, Any]
```

```
18726: ) -> ScoredResult:  
18727:     """  
18728:         FunciÃ³n de conveniencia para scoring de una pregunta.  
18729:  
18730:     Args:  
18731:         question_id: ID de pregunta  
18732:         question_global: NÃºmero global  
18733:         modality_str: String de modalidad ("TYPE_A", "TYPE_B", etc.)  
18734:         evidence_dict: Diccionario con evidencia  
18735:  
18736:     Returns:  
18737:         ScoredResult  
18738:     """  
18739:     # Parsear modalidad  
18740:     modality = ScoringModality(modality_str)  
18741:  
18742:     # Construir Evidence  
18743:     evidence = Evidence(  
18744:         elements_found=evidence_dict.get('elements_found', []),
18745:         confidence_scores=evidence_dict.get('confidence_scores', []),
18746:         semantic_similarity=evidence_dict.get('semantic_similarity'),
18747:         pattern_matches=evidence_dict.get('pattern_matches', {}),
18748:         metadata=evidence_dict.get('metadata', {})
18749:     )
18750:  
18751:     # Aplicar scoring
18752:     scorer = MicroQuestionScorer()
18753:     result = scorer.apply_scoring_modality(
18754:         question_id=question_id,
18755:         question_global=question_global,
18756:         modality=modality,
18757:         evidence=evidence
18758:     )
18759:
18760:     return result
18761:  
18762: # =====
18763: # EJEMPLO DE USO
18764: # =====
18765:  
18766: # Note: Main entry point and examples removed to maintain I/O boundary separation.
18767:     print("=*80)
18768:
18769:
18770:
18771: =====
18772: FILE: src/farfan_pipeline/analysis/spc_causal_bridge.py
18773: =====
18774:
18775: """
18776: SPC to TeoriaCambio Bridge - Causal Graph Construction.
18777:
18778: This module bridges Smart Policy Chunks (SPC) chunk graphs to causal DAG
18779: representations for integration with TeoriaCambio (Theory of Change) analysis.
18780: """
18781:
```

```
18782: from __future__ import annotations
18783:
18784: import logging
18785: from typing import Any
18786:
18787: from farfan_pipeline.core.calibration.decorators import calibrated_method
18788: from farfan_pipeline.core.parameters import ParameterLoaderV2
18789:
18790: try:
18791:     import networkx as nx
18792:
18793:     HAS_NETWORKX = True
18794: except ImportError:
18795:     HAS_NETWORKX = False
18796:     nx = None # type: ignore
18797:
18798: try:
18799:     from farfan_pipeline.processing.models import CanonPolicyPackage, ChunkGraph
18800:
18801:     HAS_CPP_MODELS = True
18802: except ImportError:
18803:     HAS_CPP_MODELS = False
18804:     CanonPolicyPackage = None # type: ignore
18805:     ChunkGraph = None # type: ignore
18806:
18807: logger = logging.getLogger(__name__)
18808:
18809:
18810: class SPCCausalBridge:
18811:     """
18812:         Converts SPC chunk graph to causal DAG for Theory of Change analysis.
18813:
18814:         This bridge enables causal analysis by mapping semantic chunk relationships
18815:         (sequential, hierarchical, reference, dependency) to causal weights that
18816:         can be used by downstream causal inference methods.
18817:     """
18818:
18819:     # Mapping of SPC edge types to causal weights
18820:     # Higher weight = stronger causal relationship
18821:     CAUSAL_WEIGHTS: dict[str, float] = {
18822:         "sequential": 0.3, # Weak temporal causality (A then B)
18823:         "hierarchical": 0.7, # Strong structural causality (A contains/governs B)
18824:         "reference": 0.5, # Medium evidential causality (A references B)
18825:         "dependency": 0.9, # Strong logical causality (A requires B)
18826:     }
18827:
18828:     def __init__(self) -> None:
18829:         """Initialize the SPC causal bridge."""
18830:         if not HAS_NETWORKX:
18831:             logger.warning(
18832:                 "NetworkX not available. SPCCausalBridge will have limited functionality. "
18833:                 "Install networkx for full causal graph construction."
18834:             )
18835:         if not HAS_CPP_MODELS:
18836:             logger.warning(
18837:                 "CPP models not available. build_causal_graph_from_cpp will have limited functionality."
```

```
18838:         )
18839:
18840:     @calibrated_method(
18841:         "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge.build_causal_graph_from_cpp"
18842:     )
18843:     def build_causal_graph_from_cpp(self, cpp: Any) -> Any:
18844:         """
18845:             Convert CanonPolicyPackage to causal DAG via chunk_graph extraction.
18846:
18847:             This method serves as the Phase 1-2 adapter integration point, enabling
18848:             D4/D6 executors to perform Theory of Change analysis on the rich
18849:             semantic structure preserved in CanonPolicyPackage.
18850:
18851:         Args:
18852:             cpp: CanonPolicyPackage from Phase 1 ingestion
18853:
18854:         Returns:
18855:             NetworkX DiGraph representing causal relationships, or None if NetworkX unavailable
18856:
18857:         Raises:
18858:             ValueError: If cpp is invalid or missing chunk_graph
18859:             ImportError: If required models not available
18860:         """
18861:         if not HAS_NETWORKX:
18862:             logger.error("NetworkX required for causal graph construction")
18863:             return None
18864:
18865:         if not HAS_CPP_MODELS:
18866:             logger.error("CPP models not available for processing")
18867:             return None
18868:
18869:         if not cpp:
18870:             raise ValueError("cpp (CanonPolicyPackage) cannot be None or empty")
18871:
18872:         if not hasattr(cpp, "chunk_graph") or not cpp.chunk_graph:
18873:             raise ValueError(
18874:                 "CanonPolicyPackage must have a valid chunk_graph attribute. "
18875:                 "Ensure Phase 1 ingestion completed successfully."
18876:             )
18877:
18878:         chunk_graph = cpp.chunk_graph
18879:
18880:         if not isinstance(chunk_graph, ChunkGraph):
18881:             raise ValueError(
18882:                 f"Expected ChunkGraph instance, got {type(chunk_graph).__name__}"
18883:             )
18884:
18885:         if not chunk_graph.chunks:
18886:             logger.warning("ChunkGraph has no chunks, returning empty causal graph")
18887:             return nx.DiGraph()
18888:
18889:         logger.info(
18890:             f"Extracting causal graph from CanonPolicyPackage: "
18891:             f"{len(chunk_graph.chunks)} chunks, {len(chunk_graph.edges)} edges"
18892:
18893:     )
```

```
18894:     chunk_graph_dict = self._convert_chunk_graph_to_dict(chunk_graph)
18895:
18896:     causal_graph = self.build_causal_graph_from_spc(chunk_graph_dict)
18897:
18898:     if causal_graph is not None:
18899:         self._enhance_graph_with_cpp_metadata(causal_graph, cpp)
18900:
18901:     return causal_graph
18902:
18903: @calibrated_method(
18904:     "farfan_core.analysis.spc_causal_bridge.SPCausalBridge._convert_chunk_graph_to_dict"
18905: )
18906: def _convert_chunk_graph_to_dict(self, chunk_graph: Any) -> dict:
18907: """
18908:     Convert ChunkGraph object to dictionary format for SPC processing.
18909:
18910:     Extracts nodes and edges from the ChunkGraph dataclass structure
18911:     and converts them to the dictionary format expected by build_causal_graph_from_spc.
18912:
18913:     Args:
18914:         chunk_graph: ChunkGraph instance from CanonPolicyPackage
18915:
18916:     Returns:
18917:         Dictionary with 'nodes' and 'edges' keys containing graph structure
18918: """
18919:     nodes = []
18920:     for chunk_id, chunk in chunk_graph.chunks.items():
18921:         node = {
18922:             "id": chunk_id,
18923:             "type": (
18924:                 getattr(chunk, "resolution", "MESO").name
18925:                 if hasattr(getattr(chunk, "resolution", None), "name")
18926:                 else "MESO"
18927:             ),
18928:             "text": getattr(chunk, "text", "")[:200],
18929:             "confidence": getattr(
18930:                 chunk, "confidence", None), "layout", 1.0
18931:             ),
18932:         }
18933:
18934:         if hasattr(chunk, "policy_area_id") and chunk.policy_area_id:
18935:             node["policy_area_id"] = chunk.policy_area_id
18936:         if hasattr(chunk, "dimension_id") and chunk.dimension_id:
18937:             node["dimension_id"] = chunk.dimension_id
18938:
18939:         nodes.append(node)
18940:
18941:     edges = []
18942:     for edge_tuple in chunk_graph.edges:
18943:         if len(edge_tuple) >= 3:
18944:             source, target, relation_type = (
18945:                 edge_tuple[0],
18946:                 edge_tuple[1],
18947:                 edge_tuple[2],
18948:             )
18949:             elif len(edge_tuple) == 2:
```

```
18950:         source, target = edge_tuple[0], edge_tuple[1]
18951:         relation_type = "sequential"
18952:     else:
18953:         logger.warning(f"Malformed edge tuple: {edge_tuple}, skipping")
18954:         continue
18955:
18956:     edges.append(
18957:         {
18958:             "source": source,
18959:             "target": target,
18960:             "type": self._normalize_edge_type(relation_type),
18961:         }
18962:     )
18963:
18964:     logger.debug(f"Converted ChunkGraph: {len(nodes)} nodes, {len(edges)} edges")
18965:
18966:     return {"nodes": nodes, "edges": edges}
18967:
18968:     @calibrated_method(
18969:         "farfan_core.analysis.spc_causal_bridge.SPCausalBridge._normalize_edge_type"
18970:     )
18971:     def _normalize_edge_type(self, relation_type: str) -> str:
18972:         """
18973:             Normalize edge relation types to causal weights mapping.
18974:
18975:             Args:
18976:                 relation_type: Original relation type from ChunkGraph
18977:
18978:             Returns:
18979:                 Normalized edge type compatible with CAUSAL_WEIGHTS
18980:         """
18981:     relation_lower = str(relation_type).lower().strip()
18982:
18983:     if relation_lower in self.CAUSAL_WEIGHTS:
18984:         return relation_lower
18985:
18986:     type_mapping = {
18987:         "seq": "sequential",
18988:         "sequence": "sequential",
18989:         "hier": "hierarchical",
18990:         "hierarchy": "hierarchical",
18991:         "parent": "hierarchical",
18992:         "child": "hierarchical",
18993:         "ref": "reference",
18994:         "cite": "reference",
18995:         "citation": "reference",
18996:         "dep": "dependency",
18997:         "require": "dependency",
18998:         "requires": "dependency",
18999:         "prerequisite": "dependency",
19000:     }
19001:
19002:     return type_mapping.get(relation_lower, "sequential")
19003:
19004:     @calibrated_method(
19005:         "farfan_core.analysis.spc_causal_bridge.SPCausalBridge._enhance_graph_with_cpp_metadata"
```

```
19006:     )
19007:     def _enhance_graph_with_cpp_metadata(self, G: Any, cpp: Any) -> None:
19008:         """
19009:             Enhance causal graph with CanonPolicyPackage metadata.
19010:
19011:             Enriches nodes with quality metrics, policy manifest data, and provenance
19012:             information from the CPP, enabling richer Theory of Change analysis.
19013:
19014:             Args:
19015:                 G: NetworkX DiGraph to enhance
19016:                 cpp: CanonPolicyPackage with metadata
19017:             """
19018:             if not HAS_NETWORKX or G is None:
19019:                 return
19020:
19021:             G.graph["schema_version"] = getattr(cpp, "schema_version", "unknown")
19022:
19023:             if hasattr(cpp, "quality_metrics") and cpp.quality_metrics:
19024:                 qm = cpp.quality_metrics
19025:                 G.graph["quality_metrics"] = {
19026:                     "provenance_completeness": getattr(qm, "provenance_completeness", 0.0),
19027:                     "structural_consistency": getattr(qm, "structural_consistency", 0.0),
19028:                     "boundary_f1": getattr(qm, "boundary_f1", 0.0),
19029:                     "kpi_linkage_rate": getattr(qm, "kpi_linkage_rate", 0.0),
19030:                     "budget_consistency_score": getattr(
19031:                         qm, "budget_consistency_score", 0.0
19032:                     ),
19033:                 }
19034:
19035:             if hasattr(cpp, "policy_manifest") and cpp.policy_manifest:
19036:                 pm = cpp.policy_manifest
19037:                 G.graph["policy_manifest"] = {
19038:                     "axes": getattr(pm, "axes", []),
19039:                     "programs": getattr(pm, "programs", []),
19040:                     "projects": getattr(pm, "projects", []),
19041:                     "years": getattr(pm, "years", []),
19042:                     "territories": getattr(pm, "territories", []),
19043:                 }
19044:
19045:                 logger.debug(
19046:                     f"Enhanced causal graph with CPP metadata: {len(G.graph)} graph attributes"
19047:                 )
19048:
19049:             @calibrated_method(
19050:                 "farfan_core.analysis.spc_causal_bridge.SPCausalBridge.build_causal_graph_from_spc"
19051:             )
19052:             def build_causal_graph_from_spc(self, chunk_graph: dict) -> Any:
19053:                 """
19054:                     Convert SPC chunk graph to causal DAG.
19055:
19056:                     Args:
19057:                         chunk_graph: Dictionary with 'nodes' and 'edges' from chunk graph
19058:
19059:                     Returns:
19060:                         NetworkX DiGraph representing causal relationships, or None if NetworkX unavailable
19061:
```

```
19062:     Raises:
19063:         ValueError: If chunk_graph is invalid
19064:     """
19065:     if not HAS_NETWORKX:
19066:         logger.error("NetworkX required for causal graph construction")
19067:         return None
19068:
19069:     if not chunk_graph or not isinstance(chunk_graph, dict):
19070:         raise ValueError("chunk_graph must be a non-empty dictionary")
19071:
19072:     nodes = chunk_graph.get("nodes", [])
19073:     edges = chunk_graph.get("edges", [])
19074:
19075:     if not nodes:
19076:         logger.warning("No nodes in chunk graph, returning empty graph")
19077:         return nx.DiGraph()
19078:
19079:     # Create directed graph
19080:     G = nx.DiGraph()
19081:
19082:     # Add nodes with attributes
19083:     for node in nodes:
19084:         node_id = node.get("id")
19085:         if node_id is None:
19086:             continue
19087:
19088:         G.add_node(
19089:             f"chunk_{node_id}",
19090:             chunk_type=node.get("type", "unknown"),
19091:             text_summary=node.get("text", "")[:100], # First 100 chars
19092:             confidence=node.get(
19093:                 "confidence",
19094:                 ParameterLoaderV2.get(
19095:                     "farfan_core.analysis.spc_causal_bridge.SPCausalBridge.build_causal_graph_from_spc",
19096:                     "auto_param_L91_50",
19097:                     0.0,
19098:                 ),
19099:             ),
19100:         )
19101:
19102:     # Add edges with causal interpretation
19103:     for edge in edges:
19104:         source = edge.get("source")
19105:         target = edge.get("target")
19106:         edge_type = edge.get("type", "sequential")
19107:
19108:         if source is None or target is None:
19109:             continue
19110:
19111:         # Convert to node IDs
19112:         # Handle both string and integer IDs
19113:         if (
19114:             isinstance(source, str)
19115:             and not source.startswith("chunk_")
19116:             or isinstance(source, int)
19117:         ):
```

```

19118:             source_id = f"chunk_{source}"
19119:         else:
19120:             source_id = str(source)
19121:
19122:         if (
19123:             isinstance(target, str)
19124:             and not target.startswith("chunk_")
19125:             or isinstance(target, int)
19126:         ):
19127:             target_id = f"chunk_{target}"
19128:         else:
19129:             target_id = str(target)
19130:
19131:         # Compute causal weight
19132:         weight = self._compute_causal_weight(edge_type)
19133:
19134:         if weight > 0: # Only add edges with positive causal weight
19135:             G.add_edge(
19136:                 source_id,
19137:                 target_id,
19138:                 weight=weight,
19139:                 edge_type=edge_type,
19140:                 original_type=edge_type,
19141:             )
19142:
19143:     # Validate and clean graph
19144:     if not nx.is_directed_acyclic_graph(G):
19145:         logger.warning("Graph contains cycles, attempting to remove cycles")
19146:         G = self._remove_cycles(G)
19147:
19148:     logger.info(
19149:         f"Built causal graph: {G.number_of_nodes()} nodes, "
19150:         f"{G.number_of_edges()} edges, "
19151:         f"is_dag={nx.is_directed_acyclic_graph(G)}"
19152:     )
19153:
19154:     return G
19155:
19156:     @calibrated_method(
19157:         "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight"
19158:     )
19159:     def _compute_causal_weight(self, edge_type: str) -> float:
19160:         """
19161:             Map SPC edge type to causal weight.
19162:
19163:             Args:
19164:                 edge_type: Type of edge from SPC graph
19165:
19166:             Returns:
19167:                 Causal weight between ParameterLoaderV2.get("farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight", "auto_param_L149_34
", 0.0) and ParameterLoaderV2.get("farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight", "auto_param_L149_42", 1.0)
19168:         """
19169:         return self.CAUSAL_WEIGHTS.get(
19170:             edge_type,
19171:             ParameterLoaderV2.get(
19172:                 "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._compute_causal_weight",

```

```
19173:             "auto_param_L151_50",
19174:             0.0,
19175:         ),
19176:     )
19177:
19178:     @calibrated_method(
19179:         "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._remove_cycles"
19180:     )
19181:     def _remove_cycles(self, G: Any) -> Any:
19182:         """
19183:             Remove cycles from graph to create a DAG.
19184:
19185:             Uses a simple strategy: remove edges with lowest weight until acyclic.
19186:
19187:             Args:
19188:                 G: NetworkX DiGraph
19189:
19190:             Returns:
19191:                 Modified graph (DAG)
19192:             """
19193:             if not HAS_NETWORKXX:
19194:                 return G
19195:
19196:             # Make a copy to avoid modifying original
19197:             G_dag = G.copy()
19198:
19199:             # Find cycles and remove lowest-weight edges
19200:             while not nx.is_directed_acyclic_graph(G_dag):
19201:                 try:
19202:                     # Find a cycle
19203:                     cycle = nx.find_cycle(G_dag, orientation="original")
19204:
19205:                     # Find edge in cycle with minimum weight
19206:                     min_weight = float("inf")
19207:                     min_edge = None
19208:
19209:                     for u, v, direction in cycle:
19210:                         if direction == "forward":
19211:                             weight = G_dag[u][v].get(
19212:                                 "weight",
19213:                                 ParameterLoaderV2.get(
19214:                                     "farfan_core.analysis.spc_causal_bridge.SPCCausalBridge._remove_cycles",
19215:                                     "auto_param_L184_59",
19216:                                     0.0,
19217:                                 ),
19218:                             )
19219:                             if weight < min_weight:
19220:                                 min_weight = weight
19221:                                 min_edge = (u, v)
19222:
19223:                     # Remove the edge
19224:                     if min_edge:
19225:                         logger.info(
19226:                             f"Removing edge {min_edge} (weight={min_weight}) to break cycle"
19227:                         )
19228:                         G_dag.remove_edge(*min_edge)
```

```
19229:         else:
19230:             # Shouldn't happen, but break to avoid infinite loop
19231:             logger.error("Could not find edge to remove from cycle")
19232:             break
19233:
19234:     except nx.NetworkXNoCycle:
19235:         # No more cycles
19236:         break
19237:
19238:     return G_dag
19239:
19240: @calibrated_method(
19241:     "farfan_core.analysis.spc_causal_bridge.SPCausalBridge.enhance_graph_with_content"
19242: )
19243: def enhance_graph_with_content(self, G: Any, chunks: list) -> Any:
19244:     """
19245:     Enhance causal graph with content-based relationships.
19246:
19247:     This method can add additional edges based on content similarity,
19248:     shared entities, or other semantic relationships.
19249:
19250:     Args:
19251:         G: NetworkX DiGraph (causal graph)
19252:         chunks: List of ChunkData objects
19253:
19254:     Returns:
19255:         Enhanced graph
19256:     """
19257:     if not HAS_NETWORKX or G is None:
19258:         return G
19259:
19260:     _ = chunks # Future enhancement: Add content-based edges using chunks
19261:     return G
19262:
19263:
19264:
19265: =====
19266: FILE: src/farfan_pipeline/analysis/teoria_cambio.py
19267: =====
19268:
19269: #!/usr/bin/env python3
19270: """
19271: Framework Unificado para la ValidaciÃ³n Causal de PolÃ-ticas PÃ³blicas
19272: =====
19273:
19274: Este script consolida un conjunto de herramientas de nivel industrial en un
19275: framework cohesivo, diseÃ±ado para la validaciÃ³n rigurosa de teorÃ-as de cambio
19276: y modelos causales (DAGs). Su propÃ³sito es servir como el motor de anÃ;lisis
19277: estructural y estocÃ;stico dentro de un flujo canÃ³nico de evaluaciÃ³n de planes
19278: de desarrollo, garantizando que las polÃ-ticas pÃ³blicas no solo sean lÃ³gicamente
19279: coherentes, sino tambiÃ©n estadÃ;sticamente robustas.
19280:
19281: Arquitectura de Vanguardia:
19282: -----
19283: 1. **Motor AxiomÃ;tico de TeorÃ-a de Cambio ('TeoriaCambio'):**
19284:     Valida la adherencia de un modelo a una jerarquÃ;a causal predefinida
```

```
19285:     (Insumos à\206\222 Procesos à\206\222 Productos à\206\222 Resultados à\206\222 Causalidad), reflejando las
19286:     dimensiones de evaluaciÃ³n (D1-D6) del flujo canÃ³nico.
19287:
19288: 2. **Validador EstocÃ;stico Avanzado ('AdvancedDAGValidator'):**
19289:     Somete los modelos causales a un escrutinio probabilÃ;stico mediante
19290:     simulaciones Monte Carlo deterministas. EvalÃ;a la aciclicidad, la
19291:     robustez estructural y el poder estadÃ;stico de la teorÃ;a.
19292:
19293: 3. **Orquestador de CertificaciÃ³n Industrial ('IndustrialGradeValidator'):**
19294:     Audita el rendimiento y la correctitud de la implementaciÃ³n del motor
19295:     axiomÃ;tico, asegurando que la herramienta de validaciÃ³n misma cumple con
19296:     estÃ;ndares de producciÃ³n.
19297:
19298: 4. **Interfaz de LÃ;nea de Comandos (CLI):**
19299:     Expone la funcionalidad a travÃ;s de una CLI robusta, permitiendo su
19300:     integraciÃ³n en flujos de trabajo automatizados y su uso como herramienta
19301:     de anÃ;lisis configurable.
19302:
19303: Autor: Sistema de ValidaciÃ³n de Planes de Desarrollo
19304: VersiÃ³n: 4.0.0 (Refactorizada y Alineada)
19305: Python: 3.10+
19306: """
19307:
19308: # =====
19309: # 1. IMPORTS Y CONFIGURACIÃ\223N GLOBAL
19310: # =====
19311:
19312: import argparse
19313: import hashlib
19314: import json
19315: import logging
19316: import random
19317: import sys
19318: import time
19319: from collections import defaultdict, deque
19320: from dataclasses import dataclass, field
19321: from datetime import datetime
19322: from enum import Enum, auto
19323: from functools import lru_cache
19324: from pathlib import Path
19325: from typing import Any, ClassVar, Optional
19326:
19327: # --- Dependencias de Terceros ---
19328: import networkx as nx
19329: import numpy as np
19330: from scipy import stats
19331:
19332: try:
19333:     from jsonschema import Draft7Validator
19334: except ImportError: # pragma: no cover - jsonschema es opcional
19335:     Draft7Validator = None
19336:
19337: # CategoriaCausal moved to farfan_core.core.types to break architectural dependency
19338: # (core.orchestrator was importing from analysis, which violates layer rules)
19339: from farfan_pipeline.core.calibration.decorators import calibrated_method
19340: from farfan_pipeline.core.parameters import ParameterLoaderV2
```

```
19341: from farfan_pipeline.core.types importCategoriaCausal
19342:
19343:
19344: # --- ConfiguraciÃ³n de Logging ---
19345: def configure_logging() -> None:
19346:     """Configura un sistema de logging de alto rendimiento para la salida estÃ;ndar."""
19347:     logging.basicConfig(
19348:         level=logging.INFO,
19349:         format="%(asctime)s.% (msecs)03d | %(levelname)-8s | %(name)s:%(lineno)d - %(message)s",
19350:         datefmt="%Y-%m-%d %H:%M:%S",
19351:         stream=sys.stdout,
19352:     )
19353:
19354:
19355: configure_logging()
19356: LOGGER = logging.getLogger(__name__)
19357:
19358: # --- Constantes Globales ---
19359: SEED: int = 42
19360: STATUS_PASSED = "â\234\205 PASÃ\223"
19361:
19362: # =====
19363: # 2. ENUMS Y ESTRUCTURAS DE DATOS (DATACLASSES)
19364: # =====
19365:
19366:
19367: class GraphType(Enum):
19368:     """TipologÃ;a de grafos para la aplicaciÃ³n de anÃ;lisis especializados."""
19369:
19370:     CAUSAL_DAG = auto()
19371:     BAYESIAN_NETWORK = auto()
19372:     STRUCTURAL_MODEL = auto()
19373:     THEORY_OF_CHANGE = auto()
19374:
19375:
19376: @dataclass
19377: class ValidacionResultado:
19378:     """Encapsula el resultado de la validaciÃ³n estructural de una teorÃ;a de cambio."""
19379:
19380:     es_valida: bool = False
19381:     violaciones_orden: list[tuple[str, str]] = field(default_factory=list)
19382:     caminos_completos: list[list[str]] = field(default_factory=list)
19383:     categorias_faltantes: list[CategoriaCausal] = field(default_factory=list)
19384:     sugerencias: list[str] = field(default_factory=list)
19385:
19386:
19387: @dataclass
19388: class ValidationMetric:
19389:     """Define una mÃ©trica de validaciÃ³n con umbrales y ponderaciÃ³n."""
19390:
19391:     name: str
19392:     value: float
19393:     unit: str
19394:     threshold: float
19395:     status: str
19396:     weight: float = 1.0
```

```
19397:  
19398:  
19399: @dataclass  
19400: class AdvancedGraphNode:  
19401:     """Nodo de grafo enriquecido con metadatos y rol semántico."""  
19402:  
19403:     name: str  
19404:     dependencies: set[str] = field(default_factory=set)  
19405:     metadata: dict[str, Any] = field(default_factory=dict)  
19406:     role: str = "variable"  
19407:  
19408:     ALLOWED_ROLES: ClassVar[set[str]] = {  
19409:         "variable",  
19410:         "insumo",  
19411:         "proceso",  
19412:         "producto",  
19413:         "resultado",  
19414:         "causalidad",  
19415:     }  
19416:  
19417:     def __post_init__(self) -> None:  
19418:         """Inicializa metadatos por defecto si no son provistos."""  
19419:         self.name = str(self.name).strip()  
19420:         if not self.name:  
19421:             raise ValueError("AdvancedGraphNode.name must be a non-empty string")  
19422:  
19423:         if not isinstance(self.dependencies, set):  
19424:             self.dependencies = set(self.dependencies or set())  
19425:         self.dependencies = {  
19426:             str(dep).strip() for dep in self.dependencies if str(dep).strip()  
19427:         }  
19428:  
19429:         self.metadata = self._normalize_metadata(self.metadata)  
19430:  
19431:         normalized_role = (self.role or "variable").strip().lower()  
19432:         if normalized_role not in self.ALLOWED_ROLES:  
19433:             raise ValueError(  
19434:                 "Invalid role '{}'. Expected one of: {}".format(  
19435:                     self.role, ", ".join(sorted(self.ALLOWED_ROLES))  
19436:                 )  
19437:             )  
19438:         self.role = normalized_role  
19439:  
19440:     def _normalize_metadata(  
19441:         self, metadata: dict[str, Any] | None = None  
19442:     ) -> dict[str, Any]:  
19443:         """Normaliza metadatos garantizando primitivos JSON y valores por defecto."""  
19444:  
19445:         source_metadata = metadata if metadata is not None else self.metadata  
19446:         base_metadata = dict(source_metadata or {})  
19447:         if not base_metadata.get("created"):  
19448:             base_metadata["created"] = datetime.now().isoformat()  
19449:         if "confidence" not in base_metadata or base_metadata["confidence"] is None:  
19450:             base_metadata["confidence"] = ParameterLoaderV2.get(  
19451:                 "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",  
19452:                 "auto_param_L174_42",
```

```
19453:         1.0,
19454:     )
19455:
19456:     normalized: dict[str, Any] = {}
19457:     for key, value in base_metadata.items():
19458:         if key == "confidence":
19459:             normalized[key] = self._sanitize_confidence(value)
19460:         elif key == "created":
19461:             normalized[key] = self._sanitize_created(value)
19462:         else:
19463:             normalized[key] = self._sanitize_metadata_value(value)
19464:     return normalized
19465:
19466:     @staticmethod
19467:     def _sanitize_confidence(value: Any) -> float:
19468:         try:
19469:             numeric = float(value)
19470:         except (TypeError, ValueError):
19471:             numeric = ParameterLoaderV2.get(
19472:                 "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
19473:                 "numeric",
19474:                 1.0,
19475:             ) # Refactored
19476:         return max(
19477:             ParameterLoaderV2.get(
19478:                 "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
19479:                 "auto_param_L192_19",
19480:                 0.0,
19481:             ),
19482:             min(
19483:                 ParameterLoaderV2.get(
19484:                     "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.__post_init__",
19485:                     "auto_param_L192_28",
19486:                     1.0,
19487:                 ),
19488:                 numeric,
19489:             ),
19490:         )
19491:
19492:     @staticmethod
19493:     def _sanitize_created(value: Any) -> str:
19494:         if isinstance(value, str) and value:
19495:             return value
19496:         if hasattr(value, "isoformat"):
19497:             try:
19498:                 return value.isoformat()
19499:             except Exception: # pragma: no cover - fallback defensivo
19500:                 pass
19501:         return datetime.now().isoformat()
19502:
19503:     @staticmethod
19504:     def _sanitize_metadata_value(value: Any) -> Any:
19505:         if isinstance(value, (str, int, float, bool)) or value is None:
19506:             return value
19507:         if hasattr(value, "isoformat"):
19508:             try:
```

```
19509:         return value.isoformat()
19510:     except Exception: # pragma: no cover - fallback defensivo
19511:         pass
19512:     return str(value)
19513:
19514:     @calibrated_method(
19515:         "farfan_core.analysis.teoria_cambio.AdvancedGraphNode.to_serializable_dict"
19516:     )
19517:     def to_serializable_dict(self) -> dict[str, Any]:
19518:         """Convierte el nodo en un diccionario serializable compatible con JSON Schema."""
19519:
19520:         metadata = self._normalize_metadata()
19521:         return {
19522:             "name": self.name,
19523:             "dependencies": sorted(self.dependencies),
19524:             "metadata": metadata,
19525:             "role": self.role,
19526:         }
19527:
19528:
19529: @dataclass
19530: class MonteCarloAdvancedResult:
19531:     """
19532:     Resultado exhaustivo de una simulaciÃ³n Monte Carlo.
19533:
19534:     Audit Point 1.1: Deterministic Seeding (RNG)
19535:     Field 'reproducible' confirms that seed was deterministically generated
19536:     and results can be reproduced with identical inputs.
19537:
19538:
19539:     plan_name: str
19540:     seed: int # Audit 1.1: Deterministic seed from _create_advanced_seed
19541:     timestamp: str
19542:     total_iterations: int
19543:     acyclic_count: int
19544:     p_value: float
19545:     bayesian_posterior: float
19546:     confidence_interval: tuple[float, float]
19547:     statistical_power: float
19548:     edge_sensitivity: dict[str, float]
19549:     node_importance: dict[str, float]
19550:     robustness_score: float
19551:     reproducible: bool # Audit 1.1: True when deterministic seed used
19552:     convergence_achieved: bool
19553:     adequate_power: bool
19554:     computation_time: float
19555:     graph_statistics: dict[str, Any]
19556:     test_parameters: dict[str, Any]
19557:
19558:
19559: # =====
19560: # 3. MOTOR AXIOMÃ201TICO DE TEORÃ215A DE CAMBIO
19561: # =====
19562:
19563:
19564: class TeoriaCambio:
```

```

19565: """
19566: Motor para la construcciÃ³n y validaciÃ³n estructural de teorÃ-as de cambio.
19567: Valida la coherencia lÃ³gica de grafos causales contra un modelo axiomÃ;tico
19568: de categorÃ-as jerÃ;rquicas, crucial para el anÃ;lisis de polÃ-ticas pÃ³blicas.
19569: """
19570:
19571: _MATRIZ_VALIDACION: dict[CategorÃaCausal, frozenset[CategorÃaCausal]] = {
19572:     cat: (
19573:         frozenset({cat, CategorÃaCausal(cat.value + 1)})
19574:         if cat.value < 5
19575:         else frozenset({cat})
19576:     )
19577:     for cat in CategorÃaCausal
19578: }
19579:
19580: def __init__(self) -> None:
19581:     """Inicializa el motor con un sistema de cache optimizado."""
19582:     self._grafo_cache: nx.DiGraph | None = None
19583:     self._cache_valido: bool = False
19584:     self.logger: logging.Logger = LOGGER
19585:
19586: @staticmethod
19587: def _esConexionValida(origen: CategorÃaCausal, destino: CategorÃaCausal) -> bool:
19588:     """Verifica la validez de una conexiÃ³n causal segÃ³n la jerarquÃ-a estructural."""
19589:     return destino in TeoriaCambio._MATRIZ_VALIDACION.get(origen, frozenset())
19590:
19591: @lru_cache(maxsize=128)
19592: @calibrated_method(
19593:     "farfan_core.analysis.teoria_cambio.TeorÃaCambio.construir_grafo_causal"
19594: )
19595: def construir_grafo_causal(self) -> nx.DiGraph:
19596:     """Construye y cachea el grafo causal canÃ³nico."""
19597:     if self._grafo_cache is not None and self._cache_valido:
19598:         self.logger.debug("Recuperando grafo causal desde cachÃ©.")
19599:         return self._grafo_cache
19600:
19601:     grafo = nx.DiGraph()
19602:     for cat in CategorÃaCausal:
19603:         grafo.add_node(cat.name, categoria=cat, nivel=cat.value)
19604:     for origen in CategorÃaCausal:
19605:         for destino in self._MATRIZ_VALIDACION.get(origen, frozenset()):
19606:             if origen != destino:
19607:                 grafo.add_edge(
19608:                     origen.name,
19609:                     destino.name,
19610:                     peso=ParameterLoaderV2.get(
19611:                         "farfan_core.analysis.teoria_cambio.TeorÃaCambio.construir_grafo_causal",
19612:                         "auto_param_L302_67",
19613:                         1.0,
19614:                     ),
19615:                 )
19616:
19617:     self._grafo_cache = grafo
19618:     self._cache_valido = True
19619:     self.logger.info(
19620:         "Grafo causal canÃ³nico construido: %d nodos, %d aristas.",
```

```
19621:         grafo.number_of_nodes(),
19622:         grafo.number_of_edges(),
19623:     )
19624:     return grafo
19625:
19626:     @calibrated_method(
19627:         "farfan_core.analysis.teoria_cambio.TeoríaCambio.construir_grafo_from_cpp"
19628:     )
19629:     def construir_grafo_from_cpp(self, cpp) -> nx.DiGraph:
19630:         """
19631:             Construir grafo causal desde CanonPolicyPackage (Phase 1 output).
19632:
19633:             Este mÃ©todo integra el Phase 1-2 adapter, permitiendo construir grafos
19634:             causales directamente desde el CanonPolicyPackage para anÃ;lisis de
19635:             TeorÃ-a de Cambio en dimensiones D4 y D6.
19636:
19637:             Args:
19638:                 cpp: CanonPolicyPackage from Phase 1 ingestion
19639:
19640:             Returns:
19641:                 NetworkX DiGraph con relaciones causales derivadas de chunk_graph
19642:
19643:             Raises:
19644:                 ValueError: If cpp is invalid
19645:             """
19646:             try:
19647:                 from farfan_pipeline.analysis.spc_causal_bridge import SPCCausalBridge
19648:
19649:                 bridge = SPCCausalBridge()
19650:                 causal_graph = bridge.build_causal_graph_from_cpp(cpp)
19651:
19652:                 if causal_graph is None:
19653:                     self.logger.warning(
19654:                         "Failed to build causal graph from CPP, using standard graph"
19655:                     )
19656:                 return self.construir_grafo_causal()
19657:
19658:                 self.logger.info(
19659:                     "Grafo causal construido desde CPP: %d nodos, %d aristas.",
19660:                     causal_graph.number_of_nodes(),
19661:                     causal_graph.number_of_edges(),
19662:                 )
19663:
19664:                 return causal_graph
19665:
19666:             except (ImportError, ValueError) as e:
19667:                 self.logger.error(f"Error building causal graph from CPP: {e}")
19668:                 return self.construir_grafo_causal()
19669:
19670:     @calibrated_method(
19671:         "farfan_core.analysis.teoria_cambio.TeoríaCambio.construir_grafo_from_spc"
19672:     )
19673:     def construir_grafo_from_spc(self, preprocessed_doc) -> nx.DiGraph:
19674:         """
19675:             Construir grafo causal desde estructura SPC (Smart Policy Chunks).
19676:
```

```
19677:     Este mÃ©todo permite construir grafos causales a partir de la estructura
19678:     semÃ¡ntica preservada por SPC, en lugar de extraer relaciones causales
19679:     Ã³nicamente del texto.
19680:
19681:     Args:
19682:         preprocessed_doc: PreprocessedDocument con modo chunked
19683:
19684:     Returns:
19685:         NetworkX DiGraph con relaciones causales derivadas de SPC
19686:     """
19687:     # Check if document is in chunked mode
19688:     if getattr(preprocessed_doc, "processing_mode", "flat") != "chunked":
19689:         # Fallback to text-based construction for flat mode
19690:         self.logger.warning(
19691:             "Document not in chunked mode, using standard causal graph"
19692:         )
19693:         return self.construir_grafo_causal()
19694:
19695:     try:
19696:         from farfan_pipeline.analysis.spc_causal_bridge import SPCCausalBridge
19697:
19698:         # Use SPC bridge to construct base graph
19699:         bridge = SPCCausalBridge()
19700:         chunk_graph = getattr(preprocessed_doc, "chunk_graph", {})
19701:
19702:         if not chunk_graph:
19703:             self.logger.warning(
19704:                 "No chunk graph available, using standard causal graph"
19705:             )
19706:             return self.construir_grafo_causal()
19707:
19708:         base_graph = bridge.build_causal_graph_from_spc(chunk_graph)
19709:
19710:         if base_graph is None:
19711:             self.logger.warning(
19712:                 "Failed to build SPC graph, using standard causal graph"
19713:             )
19714:             return self.construir_grafo_causal()
19715:
19716:         # Enhance with content analysis from chunks
19717:         chunks = getattr(preprocessed_doc, "chunks", [])
19718:         if chunks:
19719:             base_graph = bridge.enhance_graph_with_content(base_graph, chunks)
19720:
19721:             self.logger.info(
19722:                 "Grafo causal SPC construido: %d nodos, %d aristas.",
19723:                 base_graph.number_of_nodes(),
19724:                 base_graph.number_of_edges(),
19725:             )
19726:
19727:             return base_graph
19728:
19729:     except ImportError as e:
19730:         self.logger.error(f"SPCCausalBridge not available: {e}")
19731:         return self.construir_grafo_causal()
19732:
```

```
19733:     @calibrated_method(
19734:         "farfan_core.analysis.teoria_cambio.TeoríaCambio.validacion_completa"
19735:     )
19736:     def validacion_completa(self, grafo: nx.DiGraph) -> ValidacionResultado:
19737:         """Ejecuta una validaciÃ³n estructural exhaustiva de la teorÃ-a de cambio."""
19738:         resultado = ValidacionResultado()
19739:         categorias_presentes = self._extraer_categorias(grafo)
19740:         resultado.categorias_faltantes = [
19741:             c for c in CategoríaCausal if c.name not in categorias_presentes
19742:         ]
19743:         resultado.violaciones_orden = self._validar_orden_causal(grafo)
19744:         resultado.camino_completos = self._encontrar_camino_completos(grafo)
19745:         resultado.es_valida = not (
19746:             resultado.categorias_faltantes or resultado.violaciones_orden
19747:         ) and bool(resultado.camino_completos)
19748:         resultado.sugerencias = self._generar_sugerencias_internas(resultado)
19749:         return resultado
19750:
19751:     @staticmethod
19752:     def _extraer_categorias(grafo: nx.DiGraph) -> set[str]:
19753:         """Extrae el conjunto de categorÃ-as presentes en el grafo."""
19754:         return {
19755:             data["categoria"].name
19756:             for _, data in grafo.nodes(data=True)
19757:             if "categoria" in data
19758:         }
19759:
19760:     @staticmethod
19761:     def _validar_orden_causal(grafo: nx.DiGraph) -> list[tuple[str, str]]:
19762:         """Identifica las aristas que violan el orden causal axiomÃ;tico."""
19763:         violaciones = []
19764:         for u, v in grafo.edges():
19765:             cat_u = grafo.nodes[u].get("categoria")
19766:             cat_v = grafo.nodes[v].get("categoria")
19767:             if cat_u and cat_v and not TeoriaCambio._es_conexion_valida(cat_u, cat_v):
19768:                 violaciones.append((u, v))
19769:         return violaciones
19770:
19771:     @staticmethod
19772:     def _encontrar_camino_completos(grafo: nx.DiGraph) -> list[list[str]]:
19773:         """Encuentra todos los caminos simples desde nodos INSUMOS a CAUSALIDAD."""
19774:         try:
19775:             nodos_inicio = [
19776:                 n
19777:                 for n, d in grafo.nodes(data=True)
19778:                 if d.get("categoria") == CategoríaCausal.INSUMOS
19779:             ]
19780:             nodos_fin = [
19781:                 n
19782:                 for n, d in grafo.nodes(data=True)
19783:                 if d.get("categoria") == CategoríaCausal.CAUSALIDAD
19784:             ]
19785:             return [
19786:                 path
19787:                 for u in nodos_inicio
19788:                 for v in nodos_fin
```

```
19789:         for path in nx.all_simple_paths(grafo, u, v)
19790:     ]
19791: except Exception as e:
19792:     LOGGER.warning("Fallo en la detecciÃ³n de caminos completos: %s", e)
19793:     return []
19794:
19795: @staticmethod
19796: def _generar_sugerencias_internas(validacion: ValidacionResultado) -> list[str]:
19797:     """Genera un listado de sugerencias accionables basadas en los resultados."""
19798:     sugerencias = []
19799:     if validacion.categorias_faltantes:
19800:         sugerencias.append(
19801:             f"Integridad estructural comprometida. Incorporar: {', '.join(c.name for c in validacion.categorias_faltantes)}.")
19802:     )
19803:     if validacion.violaciones_orden:
19804:         sugerencias.append(
19805:             f"Corregir {len(validacion.violaciones_orden)} violaciones de secuencia causal para restaurar la coherencia lÃ³gica.")
19806:     )
19807:     if not validacion.campos_completos:
19808:         sugerencias.append(
19809:             "La teorÃ-a es incompleta. Establecer al menos un camino causal de INSUMOS a CAUSALIDAD.")
19810:     )
19811:     if validacion.es_valida:
19812:         sugerencias.append(
19813:             "La teorÃ-a es estructuralmente vÃ;lida. Proceder con anÃ;lisis de robustez estocÃ;stica.")
19814:     )
19815:     return sugerencias
19816:
19817: @calibrated_method(
19818:     "farfan_core.analysis.teoria_cambio.TeoriaCambio._execute_generar_sugerencias_internas"
19819: )
19820: def _execute_generar_sugerencias_internas(
19821:     self, validacion: "ValidacionResultado"
19822: ) -> list[str]:
19823: """
19824: Execute internal suggestion generation (wrapper method).
19825:
19826: This method wraps the static _generar_sugerencias_internas method
19827: to allow it to be called via the method executor interface.
19828:
19829: Args:
19830:     validacion: Validation result object
19831:
19832: Returns:
19833:     List of actionable suggestions
19834: """
19835:     return self._generar_sugerencias_internas(validacion)
19836:
19837:
19838: # =====
19839: # 4. VALIDADOR ESTOCÃ;STICO AVANZADO DE DAGs
19840: # =====
19841:
19842:
19843: def _create_advanced_seed(plan_name: str, salt: str = "") -> int:
19844:     """
```

```
19845:     Genera una semilla determinista de alta entropía usando SHA-512.
19846:
19847:     Audit Point 1.1: Deterministic Seeding (RNG)
19848:     Global random seed generated deterministically from plan_name and optional salt.
19849:     Confirms reproducibility across numpy/torch/PyMC stochastic elements.
19850:
19851:     Args:
19852:         plan_name: Plan identifier for deterministic derivation
19853:         salt: Optional salt for sensitivity analysis (varies to bound variance)
19854:
19855:     Returns:
19856:         64-bit unsigned integer seed derived from SHA-512 hash
19857:
19858:     Quality Evidence:
19859:         Re-run pipeline twice with identical inputs/salt a\206\222 output hashes must match 100%
19860:         Achieves MMR-level determinism per Beach & Pedersen 2019
19861:     """
19862:     combined = f"{plan_name}-{salt}".encode()
19863:     hash_obj = hashlib.sha512(combined)
19864:     seed = int.from_bytes(hash_obj.digest()[:8], "big", signed=False)
19865:
19866:     # Log for audit trail
19867:     LOGGER.info(
19868:         f"[Audit 1.1] Deterministic seed: {seed} (plan={plan_name}, salt={salt})"
19869:     )
19870:
19871:     return seed
19872:
19873:
19874: class AdvancedDAGValidator:
19875:     """
19876:     Motor para la validación estocástica y análisis de sensibilidad de DAGs.
19877:     Utiliza simulaciones Monte Carlo para cuantificar la robustez y aciclicidad
19878:     de modelos causales complejos.
19879:     """
19880:
19881:     _NODE_SCHEMA_PATH: Path = (
19882:         Path(__file__).resolve().parent
19883:         / "schemas"
19884:         / "teoria_cambio"
19885:         / "advanced_graph_node.schema.json"
19886:     )
19887:     _NODE_VALIDATOR: Any | None = None
19888:     _NODE_VALIDATION_WARNING_EMITTED: bool = False
19889:
19890:     def __init__(self, graph_type: GraphType = GraphType.CAUSAL_DAG) -> None:
19891:         self.graph_nodes: dict[str, AdvancedGraphNode] = {}
19892:         self.graph_type: GraphType = graph_type
19893:         self._rng: random.Random | None = None
19894:         self.config: dict[str, Any] = {
19895:             "default_iterations": 10000,
19896:             "confidence_level": ParameterLoaderV2.get(
19897:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.__init__",
19898:                 "auto_param_L517_32",
19899:                 0.95,
19900:             ),
19901:         },
```

```

19901:         "power_threshold": ParameterLoaderV2.get(
19902:             "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.__init__",
19903:             "auto_param_L518_31",
19904:             0.8,
19905:         ),
19906:         "convergence_threshold": 1e-5,
19907:     }
19908:     self._last_serialized_nodes: list[dict[str, Any]] = []
19909:
19910:     def add_node(
19911:         self,
19912:         name: str,
19913:         dependencies: set[str] | None = None,
19914:         role: str = "variable",
19915:         metadata: dict[str, Any] | None = None,
19916:     ) -> None:
19917:         """Agrega un nodo enriquecido al grafo."""
19918:         self.graph_nodes[name] = AdvancedGraphNode(
19919:             name, dependencies or set(), metadata or {}, role
19920:         )
19921:
19922:     @calibrated_method(
19923:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.add_edge"
19924:     )
19925:     def add_edge(self, from_node: str, to_node: str, weight: float = 1.0) -> None:
19926:         """Agrega una arista dirigida con peso opcional."""
19927:         if to_node not in self.graph_nodes:
19928:             self.add_node(to_node)
19929:         if from_node not in self.graph_nodes:
19930:             self.add_node(from_node)
19931:         self.graph_nodes[to_node].dependencies.add(from_node)
19932:         self.graph_nodes[to_node].metadata[f"edge_{from_node}>{to_node}"] = weight
19933:
19934:     @calibrated_method(
19935:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._initialize_rng"
19936:     )
19937:     def _initialize_rng(self, plan_name: str, salt: str = "") -> int:
19938:         """
19939:             Inicializa el generador de nÃºmeros aleatorios con una semilla determinista.
19940:
19941:             Audit Point 1.1: Deterministic Seeding (RNG)
19942:             Initializes numpy/random RNG with deterministic seed for reproducibility.
19943:             Sets reproducible=True in MonteCarloAdvancedResult.
19944:
19945:             Args:
19946:                 plan_name: Plan identifier for seed derivation
19947:                 salt: Optional salt for sensitivity analysis
19948:
19949:             Returns:
19950:                 Generated seed value for audit logging
19951:             """
19952:             seed = _create_advanced_seed(plan_name, salt)
19953:             self._rng = random.Random(seed)
19954:             np.random.seed(seed % (2**32))
19955:
19956:             # Log initialization for reproducibility verification

```

```

19957:         LOGGER.info(
19958:             f"[Audit 1.1] RNG initialized with seed={seed} for plan={plan_name}"
19959:         )
19960:
19961:     return seed
19962:
19963:     @staticmethod
19964:     def _is_acyclic(nodes: dict[str, AdvancedGraphNode]) -> bool:
19965:         """DetecciÃ³n de ciclos mediante el algoritmo de Kahn (ordenaciÃ³n topolÃ³gica)."""
19966:         if not nodes:
19967:             return True
19968:         in_degree = dict.fromkeys(nodes, 0)
19969:         adjacency = defaultdict(list)
19970:         for name, node in nodes.items():
19971:             for dep in node.dependencies:
19972:                 if dep in nodes:
19973:                     adjacency[dep].append(name)
19974:                     in_degree[name] += 1
19975:
19976:         queue = deque([name for name, degree in in_degree.items() if degree == 0])
19977:         count = 0
19978:         while queue:
19979:             u = queue.popleft()
19980:             count += 1
19981:             for v in adjacency[u]:
19982:                 in_degree[v] -= 1
19983:                 if in_degree[v] == 0:
19984:                     queue.append(v)
19985:         return count == len(nodes)
19986:
19987:     @calibrated_method(
19988:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._generate_subgraph"
19989:     )
19990:     def _generate_subgraph(self) -> dict[str, AdvancedGraphNode]:
19991:         """Genera un subgrafo aleatorio del grafo principal."""
19992:         if not self.graph_nodes or self._rng is None:
19993:             return {}
19994:         node_count = len(self.graph_nodes)
19995:         subgraph_size = self._rng.randint(min(3, node_count), node_count)
19996:         selected_names = self._rng.sample(list(self.graph_nodes.keys()), subgraph_size)
19997:
19998:         subgraph = {}
19999:         selected_set = set(selected_names)
20000:         for name in selected_names:
20001:             original = self.graph_nodes[name]
20002:             subgraph[name] = AdvancedGraphNode(
20003:                 name,
20004:                 original.dependencies.intersection(selected_set),
20005:                 original.metadata.copy(),
20006:                 original.role,
20007:             )
20008:         return subgraph
20009:
20010:     def calculate_acyclicity_pvalue(
20011:         self, plan_name: str, iterations: int
20012:     ) -> MonteCarloAdvancedResult:

```

```

20013:     """Cálculo avanzado de p-value con un marco estadístico completo."""
20014:     start_time = time.time()
20015:     seed = self._initialize_rng(plan_name)
20016:     if not self.graph_nodes:
20017:         self._last_serialized_nodes = []
20018:         return self._create_empty_result()
20019:         plan_name, seed, datetime.now().isoformat()
20020:     )
20021:
20022:     acyclic_count = sum(
20023:         1 for _ in range(iterations) if self._is_acyclic(self._generate_subgraph())
20024:     )
20025:
20026:     p_value = (
20027:         acyclic_count / iterations
20028:         if iterations > 0
20029:         else ParameterLoaderV2.get(
20030:             "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._generate_subgraph",
20031:             "auto_param_L633_68",
20032:             1.0,
20033:         )
20034:     )
20035:     conf_level = self.config["confidence_level"]
20036:     ci = self._calculate_confidence_interval(acyclic_count, iterations, conf_level)
20037:     power = self._calculate_statistical_power(acyclic_count, iterations)
20038:
20039:     # Análisis de Sensibilidad (simplificado para el flujo principal)
20040:     sensitivity = self._perform_sensitivity_analysis_internal(
20041:         plan_name, p_value, min(iterations, 200)
20042:     )
20043:
20044:     self.export_nodes(validate=True)
20045:
20046:     return MonteCarloAdvancedResult(
20047:         plan_name=plan_name,
20048:         seed=seed,
20049:         timestamp=datetime.now().isoformat(),
20050:         total_iterations=iterations,
20051:         acyclic_count=acyclic_count,
20052:         p_value=p_value,
20053:         bayesian_posterior=self._calculate_bayesian_posterior(p_value),
20054:         confidence_interval=ci,
20055:         statistical_power=power,
20056:         edge_sensitivity=sensitivity.get("edge_sensitivity", {}),
20057:         node_importance=self._calculate_node_importance(),
20058:         robustness_score=1 / (1 + sensitivity.get("average_sensitivity", 0)),
20059:         reproducible=True, # La reproducibilidad es por diseño de la semilla
20060:         convergence_achieved=(p_value * (1 - p_value) / iterations)
20061:         < self.config["convergence_threshold"],
20062:         adequate_power=power >= self.config["power_threshold"],
20063:         computation_time=time.time() - start_time,
20064:         graph_statistics=self.get_graph_stats(),
20065:         test_parameters={"iterations": iterations, "confidence_level": conf_level},
20066:     )
20067:
20068:     @property

```

```

20069:     @calibrated_method(
20070:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes"
20071:     )
20072:     def last_serialized_nodes(self) -> list[dict[str, Any]]:
20073:         """Obtiene la instantánea más reciente de nodos serializados."""
20074:
20075:         return [
20076:             {
20077:                 "name": node["name"],
20078:                 "dependencies": list(node["dependencies"]),
20079:                 "metadata": dict(node["metadata"]),
20080:                 "role": node["role"],
20081:             }
20082:             for node in self._last_serialized_nodes
20083:         ]
20084:
20085:     def export_nodes(
20086:         self, validate: bool = False, schema_path: Path | None = None
20087:     ) -> list[dict[str, Any]]:
20088:         """Serializa los nodos del grafo y opcionalmente valida contra JSON Schema."""
20089:
20090:         serialized_nodes = [
20091:             node.to_serializable_dict()
20092:             for node in sorted(self.graph_nodes.values(), key=lambda n: n.name)
20093:         ]
20094:         self._last_serialized_nodes = serialized_nodes
20095:
20096:         if validate:
20097:             validator = self._get_node_validator(schema_path)
20098:             if validator is not None:
20099:                 for index, payload in enumerate(serialized_nodes):
20100:                     errors = list(validator.iter_errors(payload))
20101:                     if errors:
20102:                         joined = "; ".join(
20103:                             (
20104:                                 f"{'/'.join(str(x) for x in error.path)}: {error.message}"
20105:                                 if error.path
20106:                                 else error.message
20107:                             )
20108:                             for error in errors
20109:                         )
20110:                         raise ValueError(
20111:                             "AdvancedGraphNode payload at index %d failed schema validation: %s"
20112:                             % (index, joined)
20113:                         )
20114:
20115:             return serialized_nodes
20116:
20117:     @classmethod
20118:     def _get_node_validator(
20119:         cls, schema_path: Path | None = None
20120:     ) -> Optional["Draft7Validator"]:
20121:         """Obtiene (y cachea) el validador JSON Schema para nodos avanzados."""
20122:
20123:         if Draft7Validator is None:
20124:             if not cls._NODE_VALIDATION_WARNING_EMITTED:

```

```
20125:             LOGGER.warning(
20126:                 "jsonschema is not installed; skipping AdvancedGraphNode schema validation."
20127:             )
20128:             cls._NODE_VALIDATION_WARNING_EMITTED = True
20129:         return None
20130:
20131:     if schema_path is None and cls._NODE_VALIDATOR is not None:
20132:         return cls._NODE_VALIDATOR
20133:
20134:     path = Path(schema_path) if schema_path else cls._NODE_SCHEMA_PATH
20135:
20136:     # Delegate to factory for I/O operation
20137:     from farfan_pipeline.analysis.factory import load_json
20138:
20139:     try:
20140:         schema = load_json(path)
20141:     except FileNotFoundError:
20142:         LOGGER.error("Advanced graph node schema file not found at %s", path)
20143:         return None
20144:     except json.JSONDecodeError as exc:
20145:         LOGGER.error("Invalid JSON in advanced graph node schema %s: %s", path, exc)
20146:         return None
20147:
20148:     validator = Draft7Validator(schema)
20149:     if schema_path is None:
20150:         cls._NODE_VALIDATOR = validator
20151:     return validator
20152:
20153:     def _perform_sensitivity_analysis_internal(
20154:         self, plan_name: str, base_p_value: float, iterations: int
20155:     ) -> dict[str, Any]:
20156:         """Análisis de sensibilidad interno optimizado para evitar cálculos redundantes."""
20157:         edge_sensitivity: dict[str, float] = {}
20158:         # 1. Genera los subgrafos una sola vez
20159:         subgraphs = []
20160:         for _ in range(iterations):
20161:             subgraph = self._generate_subgraph()
20162:             subgraphs.append(subgraph)
20163:         # 2. Lista de todas las aristas
20164:         edges = {
20165:             f"{dep}>{name}"
20166:             for name, node in self.graph_nodes.items()
20167:             for dep in node.dependencies
20168:         }
20169:         # 3. Para cada arista, calcula el p-value perturbado usando los mismos subgrafos
20170:         for edge in edges:
20171:             from_node, to_node = edge.split("->")
20172:             acyclic_count = 0
20173:             for subgraph in subgraphs:
20174:                 # Perturba el subgrafo removiendo la arista
20175:                 if to_node in subgraph and from_node in subgraph[to_node].dependencies:
20176:                     subgraph_copy = {
20177:                         k: AdvancedGraphNode(
20178:                             v.name, set(v.dependencies), dict(v.metadata), v.role
20179:                         )
20180:                         for k, v in subgraph.items()
```

```

20181:         }
20182:         subgraph_copy[to_node].dependencies.discard(from_node)
20183:     else:
20184:         subgraph_copy = subgraph
20185:     if AdvancedDAGValidator._is_acyclic(subgraph_copy):
20186:         acyclic_count += 1
20187:     perturbed_p = acyclic_count / iterations
20188:     edge_sensitivity[edge] = abs(base_p_value - perturbed_p)
20189:     sens_values = list(edge_sensitivity.values())
20190:     return {
20191:         "edge_sensitivity": edge_sensitivity,
20192:         "average_sensitivity": np.mean(sens_values) if sens_values else 0,
20193:     }
20194:
20195:     @staticmethod
20196:     def _calculate_confidence_interval(
20197:         s: int, n: int, conf: float
20198:     ) -> tuple[float, float]:
20199:         """Calcula el intervalo de confianza de Wilson."""
20200:         if n == 0:
20201:             return (
20202:                 ParameterLoaderV2.get(
20203:                     "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20204:                     "auto_param_L793_20",
20205:                     0.0,
20206:                 ),
20207:                 ParameterLoaderV2.get(
20208:                     "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20209:                     "auto_param_L793_25",
20210:                     1.0,
20211:                 ),
20212:             )
20213:         z = stats.norm.ppf(1 - (1 - conf) / 2)
20214:         p_hat = s / n
20215:         den = 1 + z**2 / n
20216:         center = (p_hat + z**2 / (2 * n)) / den
20217:         width = (z * np.sqrt(p_hat * (1 - p_hat) / n + z**2 / (4 * n**2))) / den
20218:         return (max(0, center - width), min(1, center + width))
20219:
20220:     @staticmethod
20221:     def _calculate_statistical_power(
20222:         s: int,
20223:         n: int,
20224:         alpha: float = ParameterLoaderV2.get(
20225:             "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20226:             "auto_param_L802_68",
20227:             0.05,
20228:         ),
20229:     ) -> float:
20230:         """Calcula el poder estadÃ-stico a posteriori."""
20231:         if n == 0:
20232:             return ParameterLoaderV2.get(
20233:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20234:                 "auto_param_L805_19",
20235:                 0.0,
20236:             )

```

```

20237:         p = s / n
20238:         effect_size = 2 * (
20239:             np.arcsin(np.sqrt(p))
20240:             - np.arcsin(
20241:                 np.sqrt(
20242:                     ParameterLoaderV2.get(
20243:                         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20244:                         "auto_param_L807_69",
20245:                         0.5,
20246:                     )
20247:                 )
20248:             )
20249:         )
20250:         return stats.norm.sf(
20251:             stats.norm.ppf(1 - alpha) - abs(effect_size) * np.sqrt(n / 2)
20252:         )
20253:
20254:     @staticmethod
20255:     def _calculate_bayesian_posterior(
20256:         likelihood: float,
20257:         prior: float = ParameterLoaderV2.get(
20258:             "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.last_serialized_nodes",
20259:             "auto_param_L813_72",
20260:             0.5,
20261:         ),
20262:     ) -> float:
20263:         """Calcula la probabilidad posterior Bayesiana simple."""
20264:         if (likelihood * prior + (1 - likelihood) * (1 - prior)) == 0:
20265:             return prior
20266:         return (likelihood * prior) / (
20267:             likelihood * prior + (1 - likelihood) * (1 - prior)
20268:         )
20269:
20270:     @calibrated_method(
20271:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator._calculate_node_importance"
20272:     )
20273:     def _calculate_node_importance(self) -> dict[str, float]:
20274:         """Calcula una mÃ©trica de importancia para cada nodo."""
20275:         if not self.graph_nodes:
20276:             return {}
20277:         out_degree = defaultdict(int)
20278:         for node in self.graph_nodes.values():
20279:             for dep in node.dependencies:
20280:                 out_degree[dep] += 1
20281:
20282:             max_centrality = (
20283:                 max(
20284:                     len(node.dependencies) + out_degree[name]
20285:                     for name, node in self.graph_nodes.items()
20286:                 )
20287:                 or 1
20288:             )
20289:         return {
20290:             name: (len(node.dependencies) + out_degree[name]) / max_centrality
20291:             for name, node in self.graph_nodes.items()
20292:         }

```

```
20293:
20294:     @calibrated_method(
20295:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats"
20296:     )
20297:     def get_graph_stats(self) -> dict[str, Any]:
20298:         """Obtiene estadÃ-sticas estructurales del grafo."""
20299:         nodes = len(self.graph_nodes)
20300:         edges = sum(len(n.dependencies) for n in self.graph_nodes.values())
20301:         return {
20302:             "nodes": nodes,
20303:             "edges": edges,
20304:             "density": edges / (nodes * (nodes - 1)) if nodes > 1 else 0,
20305:         }
20306:
20307:     def _create_empty_result(
20308:         self, plan_name: str, seed: int, timestamp: str
20309:     ) -> MonteCarloAdvancedResult:
20310:         """Crea un resultado vacÃ-o para grafos sin nodos."""
20311:         return MonteCarloAdvancedResult(
20312:             plan_name,
20313:             seed,
20314:             timestamp,
20315:             0,
20316:             0,
20317:             ParameterLoaderV2.get(
20318:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20319:                 "auto_param_L864_12",
20320:                 1.0,
20321:             ),
20322:             ParameterLoaderV2.get(
20323:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20324:                 "auto_param_L865_12",
20325:                 1.0,
20326:             ),
20327:             (
20328:                 ParameterLoaderV2.get(
20329:                     "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20330:                     "auto_param_L866_13",
20331:                     0.0,
20332:                 ),
20333:                 ParameterLoaderV2.get(
20334:                     "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20335:                     "auto_param_L866_18",
20336:                     1.0,
20337:                 ),
20338:             ),
20339:             ParameterLoaderV2.get(
20340:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20341:                 "auto_param_L867_12",
20342:                 0.0,
20343:             ),
20344:             {},
20345:             {},
20346:             ParameterLoaderV2.get(
20347:                 "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20348:                 "auto_param_L870_12",
```

```
20349:         1.0,
20350:     ),
20351:     True,
20352:     True,
20353:     False,
20354:     ParameterLoaderV2.get(
20355:         "farfan_core.analysis.teoria_cambio.AdvancedDAGValidator.get_graph_stats",
20356:         "auto_param_L874_12",
20357:         0.0,
20358:     ),
20359:     {},
20360:     {},
20361: )
20362:
20363:
20364: # =====
20365: # 5. ORQUESTADOR DE CERTIFICACIÃ“N INDUSTRIAL
20366: # =====
20367:
20368:
20369: class IndustrialGradeValidator:
20370: """
20371:     Orquesta una validaciÃ³n de grado industrial para el motor de TeorÃ-a de Cambio.
20372: """
20373:
20374:     def __init__(self) -> None:
20375:         self.logger: logging.Logger = LOGGER
20376:         self.metrics: list[ValidationMetric] = []
20377:         self.performance_benchmarks: dict[str, float] = {
20378:             "engine_readiness": ParameterLoaderV2.get(
20379:                 "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
20380:                 "auto_param_L892_32",
20381:                 0.05,
20382:             ),
20383:             "graph_construction": ParameterLoaderV2.get(
20384:                 "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
20385:                 "auto_param_L893_34",
20386:                 0.1,
20387:             ),
20388:             "path_detection": ParameterLoaderV2.get(
20389:                 "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
20390:                 "auto_param_L894_30",
20391:                 0.2,
20392:             ),
20393:             "full_validation": ParameterLoaderV2.get(
20394:                 "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.__init__",
20395:                 "auto_param_L895_31",
20396:                 0.3,
20397:             ),
20398:         }
20399:
20400:     @calibrated_method(
20401:         "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.execute_suite"
20402:     )
20403:     def execute_suite(self) -> bool:
20404:         """Ejecuta la suite completa de validaciÃ³n industrial."""

```

```
20405:         self.logger.info("=" * 80)
20406:         self.logger.info("INICIO DE SUITE DE CERTIFICACIÃN INDUSTRIAL")
20407:         self.logger.info("=" * 80)
20408:         start_time = time.time()
20409:
20410:         results = [
20411:             self.validate_engine_readiness(),
20412:             self.validate_causal_categories(),
20413:             self.validate_connection_matrix(),
20414:             self.run_performance_benchmarks(),
20415:         ]
20416:
20417:         total_time = time.time() - start_time
20418:         passed = sum(1 for m in self.metrics if m.status == STATUS_PASSED)
20419:         success_rate = (passed / len(self.metrics) * 100) if self.metrics else 100
20420:
20421:         self.logger.info("\n" + "=" * 80)
20422:         self.logger.info("ð\237\223\212 INFORME DE CERTIFICACIÃN INDUSTRIAL")
20423:         self.logger.info("=" * 80)
20424:         self.logger.info(f" - Tiempo Total de la Suite: {total_time:.3f} segundos")
20425:         self.logger.info(
20426:             f" - Tasa de Ãxito de MÃtricas: {success_rate:.1f}%% ({passed}/{len(self.metrics)})"
20427:         )
20428:
20429:         meets_standards = all(results) and success_rate >= 90.0
20430:         self.logger.info(
20431:             f" ð\237\217\206 VEREDICTO: {'CERTIFICACIÃN OTORGADA' if meets_standards else 'SE REQUIEREN MEJORAS'}"
20432:         )
20433:         return meets_standards
20434:
20435:     @calibrated_method(
20436:         "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_engine_readiness"
20437:     )
20438:     def validate_engine_readiness(self) -> bool:
20439:         """Valida la disponibilidad y tiempo de instanciaciÃ³n de los motores de anÃ¡lisis."""
20440:         self.logger.info("[Capa 1] Validando disponibilidad de motores...")
20441:         start_time = time.time()
20442:         try:
20443:             _ = TeoriaCambio()
20444:             _ = AdvancedDAGValidator()
20445:             instantiation_time = time.time() - start_time
20446:             metric = self._log_metric(
20447:                 "Disponibilidad del Motor",
20448:                 instantiation_time,
20449:                 "s",
20450:                 self.performance_benchmarks["engine_readiness"],
20451:             )
20452:             return metric.status == STATUS_PASSED
20453:         except Exception as e:
20454:             self.logger.error("  ã\235\214 Error crÃ-tico al instanciar motores: %s", e)
20455:             return False
20456:
20457:     @calibrated_method(
20458:         "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_causal_categories"
20459:     )
20460:     def validate_causal_categories(self) -> bool:
```

```
20461:     """Valida la completitud y el orden axiomático de las categorías causales."""
20462:     self.logger.info(" [Capa 2] Validando axiomas de categorías causales...")
20463:     expected = {cat.name: cat.value for cat in CategoriaCausal}
20464:     if len(expected) != 5 or any(
20465:         expected[name] != i + 1
20466:         for i, name in enumerate(
20467:             ["INSUMOS", "PROCESOS", "PRODUCTOS", "RESULTADOS", "CAUSALIDAD"]
20468:         )
20469:     ):
20470:         self.logger.error(
20471:             "     \u235c\u2143 Definición de CategoriaCausal es inconsistente con el axioma."
20472:         )
20473:         return False
20474:     self.logger.info("     \u234c\u2055 Axiomas de categorías validados.")
20475:     return True
20476:
20477: @calibrated_method(
20478:     "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.validate_connection_matrix"
20479: )
20480: def validate_connection_matrix(self) -> bool:
20481:     """Valida la matriz de transiciones causales."""
20482:     self.logger.info(" [Capa 3] Validando matriz de transiciones causales...")
20483:     tc = TeoriaCambio()
20484:     errors = 0
20485:     for o in CategoriaCausal:
20486:         for d in CategoriaCausal:
20487:             is_valid = tc._esConexionValida(o, d)
20488:             expected = d in tc._MATRIZ_VALIDACION.get(o, set())
20489:             if is_valid != expected:
20490:                 errors += 1
20491:     if errors > 0:
20492:         self.logger.error(
20493:             "     \u235c\u2143 %d inconsistencias encontradas en la matriz de validación.",
20494:             errors,
20495:         )
20496:         return False
20497:     self.logger.info("     \u234c\u2055 Matriz de transiciones validada.")
20498:     return True
20499:
20500: @calibrated_method(
20501:     "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator.run_performance_benchmarks"
20502: )
20503: def run_performance_benchmarks(self) -> bool:
20504:     """Ejecuta benchmarks de rendimiento para las operaciones críticas del motor."""
20505:     self.logger.info(" [Capa 4] Ejecutando benchmarks de rendimiento...")
20506:     tc = TeoriaCambio()
20507:
20508:     grafo = self._benchmark_operation(
20509:         "Construcción de Grafo",
20510:         tc.construir_grafo_causal,
20511:         self.performance_benchmarks["graph_construction"],
20512:     )
20513:     _ = self._benchmark_operation(
20514:         "Detección de Caminos",
20515:         tc._encontrar_caminos_completos,
20516:         self.performance_benchmarks["path_detection"],
```

```
20517:         grafo,
20518:     )
20519:     _ = self._benchmark_operation(
20520:         "ValidaciÃ³n Completa",
20521:         tc.validacion_completa,
20522:         self.performance_benchmarks["full_validation"],
20523:         grafo,
20524:     )
20525:
20526:     return all(
20527:         m.status == STATUS_PASSED
20528:         for m in self.metrics
20529:         if m.name in self.performance_benchmarks
20530:     )
20531:
20532:     def _benchmark_operation(
20533:         self, operation_name: str, callable_obj, threshold: float, *args, **kwargs
20534:     ):
20535:         """Mide el tiempo de ejecuciÃ³n de una operaciÃ³n y registra la mÃ©trica."""
20536:         start_time = time.time()
20537:         result = callable_obj(*args, **kwargs)
20538:         elapsed = time.time() - start_time
20539:         self._log_metric(operation_name, elapsed, "s", threshold)
20540:         return result
20541:
20542:     @calibrated_method(
20543:         "farfan_core.analysis.teoria_cambio.IndustrialGradeValidator._log_metric"
20544:     )
20545:     def _log_metric(self, name: str, value: float, unit: str, threshold: float):
20546:         """Registra y reporta una mÃ©trica de validaciÃ³n."""
20547:         status = STATUS_PASSED if value <= threshold else "\u2351\u2141 FALL\u2351\u2231"
20548:         metric = ValidationMetric(name, value, unit, threshold, status)
20549:         self.metrics.append(metric)
20550:         icon = "\u2371\u2374" if status == STATUS_PASSED else "\u2371\u2241"
20551:         self.logger.info(
20552:             f" {icon} {name}: {value:.4f} {unit} (LÃ¡mite: {threshold:.4f} {unit}) - {status}"
20553:         )
20554:     return metric
20555:
20556:
20557: # =====
20558: # 6. LÃGICA DE LA CLI Y CONSTRUCTORES DE GRAFOS DE DEMOSTRACIÃN
20559: # =====
20560:
20561:
20562: def create_policy_theory_of_change_graph() -> AdvancedDAGValidator:
20563:     """
20564:         Construye un grafo causal de demostraciÃ³n alineado con la polÃ-tica P1:
20565:         "Derechos de las mujeres e igualdad de gÃ©nero".
20566:     """
20567:     validator = AdvancedDAGValidator(graph_type=GraphType.THEORY_OF_CHANGE)
20568:
20569:     # Nodos basados en el lexicÃ³n y las dimensiones D1-D5
20570:     validator.add_node("recursos_financieros", role="insumo")
20571:     validator.add_node(
20572:         "mecanismos_de_adelanto", dependencies={"recursos_financieros"}, role="proceso"
```

```
20573:     )
20574:     validator.add_node(
20575:         "comisarias_funcionales",
20576:         dependencies={"mecanismos_de_adelanto"}, 
20577:         role="producto",
20578:     )
20579:     validator.add_node(
20580:         "reduccion_vbg", dependencies={"comisarias_funcionales"}, role="resultado"
20581:     )
20582:     validator.add_node(
20583:         "aumento_participacion_politica",
20584:         dependencies={"mecanismos_de_adelanto"}, 
20585:         role="resultado",
20586:     )
20587:     validator.add_node(
20588:         "autonomia_economica",
20589:         dependencies={"reduccion_vbg", "aumento_participacion_politica"}, 
20590:         role="causalidad",
20591:     )
20592:
20593:     LOGGER.info("Grafo de demostraciÃ³n para la polÃ-tica 'P1' construido.")
20594:     return validator
20595:
20596:
20597: def main() -> None:
20598:     """Punto de entrada principal para la interfaz de lÃnea de comandos (CLI)."""
20599:     parser = argparse.ArgumentParser(
20600:         description="Framework Unificado para la ValidaciÃ³n Causal de PolÃ-ticas PÃ³blicas.",
20601:         formatter_class=argparse.RawTextHelpFormatter,
20602:     )
20603:     subparsers = parser.add_subparsers(dest="command", required=True)
20604:
20605:     # --- Comando: industrial-check ---
20606:     subparsers.add_parser(
20607:         "industrial-check",
20608:         help="Ejecuta la suite de certificaciÃ³n industrial sobre los motores de validaciÃ³n.",
20609:     )
20610:
20611:     # --- Comando: stochastic-validation ---
20612:     parser_stochastic = subparsers.add_parser(
20613:         "stochastic-validation",
20614:         help="Ejecuta la validaciÃ³n estocÃ;sica sobre un modelo causal de polÃ-tica.",
20615:     )
20616:     parser_stochastic.add_argument(
20617:         "plan_name",
20618:         type=str,
20619:         help="Nombre del plan o polÃ-tica a validar (usado como semilla).",
20620:     )
20621:     parser_stochastic.add_argument(
20622:         "-i",
20623:         "--iterations",
20624:         type=int,
20625:         default=10000,
20626:         help="NÃºmero de iteraciones para la simulaciÃ³n Monte Carlo.",
20627:     )
20628:
```

```
20629:     args = parser.parse_args()
20630:
20631:     if args.command == "industrial-check":
20632:         validator = IndustrialGradeValidator()
20633:         success = validator.execute_suite()
20634:         sys.exit(0 if success else 1)
20635:
20636:     elif args.command == "stochastic-validation":
20637:         LOGGER.info("Iniciando validaciÃ³n estocÃ¡stica para el plan: %s", args.plan_name)
20638:         # Se podrÃ¡a cargar un grafo desde un archivo, pero para la demo usamos el constructor
20639:         dag_validator = create_policy_theory_of_change_graph()
20640:         result = dag_validator.calculate_acyclicity_pvalue(
20641:             args.plan_name, args.iterations
20642:         )
20643:         serialized_nodes = dag_validator.last_serialized_nodes
20644:
20645:         LOGGER.info("\n" + "=" * 80)
20646:         LOGGER.info(
20647:             f"RESULTADOS DE LA VALIDACIÃN ESTOCÃSTICA PARA '{result.plan_name}'"
20648:         )
20649:         LOGGER.info("=" * 80)
20650:         LOGGER.info(f" - P-value (Aciclicidad): {result.p_value:.6f}")
20651:         LOGGER.info(
20652:             f" - Posterior Bayesiano de Aciclicidad: {result.bayesian_posterior:.4f}"
20653:         )
20654:         LOGGER.info(
20655:             f" - Intervalo de Confianza (95%): [{result.confidence_interval[0]:.4f}, {result.confidence_interval[1]:.4f}]"
20656:         )
20657:         LOGGER.info(
20658:             f" - Poder EstadÃ¡stico: {result.statistical_power:.4f} {'(ADECUADO)' if result.adequate_power else '(INSUFICIENTE)'}"
20659:         )
20660:         LOGGER.info(f" - Score de Robustez Estructural: {result.robustness_score:.4f}")
20661:         LOGGER.info(f" - Tiempo de CÃ³mputo: {result.computation_time:.3f}s")
20662:         LOGGER.info(f" - Nodos validados contra schema: %d", len(serialized_nodes))
20663:         LOGGER.info("=" * 80)
20664:
20665:
20666: # =====
20667: # 7. PUNTO DE ENTRADA
20668: # =====
20669:
20670:
```