```
            "dimension": dimension.value,
            "contradictions": [self._serialize_contradiction(c) for c in contradictions],
            "total_contradictions": len(contradictions),
            "high_severity_count": sum(1 for c in contradictions if c.severity > get_param
eter_loader().get("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._i
nitialize_pdm_patterns").get("auto_param_L469_81", 0.7)),
            "coherence_metrics": coherence_metrics,
            "recommendations": recommendations,
            "knowledge_graph_stats": self._get_graph_statistics()
        }

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

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

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

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

            # Determinar rol semántico
            semantic_role = self._determine_semantic_role(sent)

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

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

            statements.append(statement)

        return statements

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

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

```
                entities=stmt.entities,
                temporal_markers=stmt.temporal_markers,
                quantitative_claims=stmt.quantitative_claims,
                embedding=embedding,
                context_window=stmt.context_window,
                semantic_role=stmt.semantic_role,
                dependencies=stmt.dependencies
            )
            enhanced_statements.append(enhanced)

        return enhanced_statements

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

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

            # Conectar con declaraciones relacionadas
            for j, other in enumerate(statements):
                if i != j:
                    similarity = self._calculate_similarity(stmt, other)
                    if similarity > get_parameter_loader().get("saaaaaa.analysis.contradic
tion_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L562_3
6", 0.7):  # Umbral de relación
                        self.knowledge_graph.add_edge(
                            f"stmt_{i}",
                            f"stmt_{j}",
                            weight=similarity,
                            relation_type=self._determine_relation_type(stmt, other)
                        )

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

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

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

                    if contradiction_score > get_parameter_loader().get("saaaaaa.analysis.
contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_par
am_L587_45", 0.7) and similarity > get_parameter_loader().get("saaaaaa.analysis.contradict
ion_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L587_66
", 0.5):
                        # Calcular confianza Bayesiana
                        confidence = self.bayesian_calculator.calculate_posterior(
                            evidence_strength=contradiction_score,
                            observations=len(stmt_a.entities) + len(stmt_b.entities),
```

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

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

        return contradictions

    def _detect_numerical_inconsistencies(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta inconsistencias numéricas con análisis estadístico"""
        contradictions = []

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

                                if divergence is not None and divergence > get_parameter_l
oader().get("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._build_k
nowledge_graph").get("auto_param_L632_75", 0.2):
                                    # Test estadístico de significancia
                                    p_value = self._statistical_significance_test(
                                        claim_a,
                                        claim_b
                                    )

                                    if p_value < get_parameter_loader().get("saaaaaa.analy
sis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto
_param_L639_49", 0.05):  # Significancia estadística
                                        confidence =
self.bayesian_calculator.calculate_posterior(
                                            evidence_strength=1 - p_value,
                                            observations=2,
                                            domain_weight=1.5  # Mayor peso para evidencia
 numérica
                                        )

                                        evidence = ContradictionEvidence(
                                            statement_a=stmt_a,
                                            statement_b=stmt_b,

contradiction_type=ContradictionType.NUMERICAL_INCONSISTENCY,
                                            confidence=confidence,
                                            severity=min(get_parameter_loader().get("saaaa
aa.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").g
```

```
et("auto_param_L651_57", 1.0), divergence),
                                semantic_similarity=get_parameter_loader().get
("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_gr
aph").get("auto_param_L652_64", 0.0),
                                logical_conflict_score=divergence,
                                temporal_consistency=True,
                                numerical_divergence=divergence,
                                affected_dimensions=[stmt_a.dimension],

resolution_suggestions=self._suggest_resolutions(
                                    ContradictionType.NUMERICAL_INCONSISTENCY
                                ),
                                statistical_significance=p_value
                            )
                            contradictions.append(evidence)

        return contradictions

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

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

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

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

                confidence = self.bayesian_calculator.calculate_posterior(
                    evidence_strength=get_parameter_loader().get("saaaaaa.analysis.contrad
iction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L686
_38", 0.9),  # Alta confianza en lógica temporal
                    observations=len(conflicts),
                    domain_weight=1.2
                )

                evidence = ContradictionEvidence(
                    statement_a=stmt_a,
                    statement_b=stmt_b,
                    contradiction_type=ContradictionType.TEMPORAL_CONFLICT,
                    confidence=confidence,
                    severity=get_parameter_loader().get("saaaaaa.analysis.contradiction_de
teccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L696_29",
0.8),  # Los conflictos temporales son severos
                    semantic_similarity=self._calculate_similarity(stmt_a, stmt_b),
                    logical_conflict_score=get_parameter_loader().get("saaaaaa.analysis.co
ntradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param
_L698_43", 1.0),
                    temporal_consistency=False,
                    numerical_divergence=None,
                    affected_dimensions=[PolicyDimension.PROGRAMATICO],
                    resolution_suggestions=self._suggest_resolutions(
                        ContradictionType.TEMPORAL_CONFLICT
                    )
                )
                contradictions.append(evidence)

        return contradictions
```

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

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

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

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

                    if self._has_logical_conflict(stmt_a, stmt_b):
                        confidence = self.bayesian_calculator.calculate_posterior(
                            evidence_strength=get_parameter_loader().get("saaaaaa.analysis
.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_pa
ram_L736_46", 0.85),
                            observations=len(cycle),
                            domain_weight = get_parameter_loader().get("saaaaaa.analysis.c
ontradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("domain_we
ight", 1.0) # Refactored
                        )

                        evidence = ContradictionEvidence(
                            statement_a=stmt_a,
                            statement_b=stmt_b,
                            contradiction_type=ContradictionType.LOGICAL_INCOMPATIBILITY,
                            confidence=confidence,
                            severity=get_parameter_loader().get("saaaaaa.analysis.contradi
ction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L746_
37", 0.7),
                            semantic_similarity=self._calculate_similarity(stmt_a,
stmt_b),
                            logical_conflict_score=get_parameter_loader().get("saaaaaa.ana
lysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("au
to_param_L748_51", 0.9),
                            temporal_consistency=True,
                            numerical_divergence=None,
                            affected_dimensions=[stmt_a.dimension, stmt_b.dimension],
                            resolution_suggestions=self._suggest_resolutions(
                                ContradictionType.LOGICAL_INCOMPATIBILITY
                            ),
                            graph_path=cycle
                        )
                        contradictions.append(evidence)
        except nx.NetworkXError:
            pass  # No negative cycles found

        return contradictions

    def _detect_resource_conflicts(
            self,
            statements: list[PolicyStatement]
    ) -> list[ContradictionEvidence]:
        """Detecta conflictos en asignación de recursos"""
        contradictions = []
```

```
        resource_allocations = {}

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

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

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

                                evidence = ContradictionEvidence(
                                    statement_a=stmt_a,
                                    statement_b=stmt_b,

contradiction_type=ContradictionType.RESOURCE_ALLOCATION_MISMATCH,
                                    confidence=confidence,
                                    severity=get_parameter_loader().get("saaaaaa.analysis.
contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_par
am_L804_45", 0.9),  # Conflictos de recursos son críticos
                                    semantic_similarity=self._calculate_similarity(stmt_a,
 stmt_b),
                                    logical_conflict_score=get_parameter_loader().get("saa
aaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph")
.get("auto_param_L806_59", 0.8),
                                    temporal_consistency=True,
                                    numerical_divergence=abs(amount_a - amount_b) /
max(amount_a, amount_b),
                                    affected_dimensions=[PolicyDimension.FINANCIERO],
                                    resolution_suggestions=self._suggest_resolutions(
                                        ContradictionType.RESOURCE_ALLOCATION_MISMATCH
                                    )
                                )
                                contradictions.append(evidence)

        return contradictions

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

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

```
        # Índice de coherencia semántica global
        semantic_coherence = self._calculate_global_semantic_coherence(statements)

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

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

        # Índice de fragmentación del grafo
        graph_fragmentation = self._calculate_graph_fragmentation()

        # Score de coherencia compuesto (weighted harmonic mean)
        weights = np.array([get_parameter_loader().get("saaaaaa.analysis.contradiction_det
eccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_param_L845_28",
0.3), get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.PolicyContradic
tionDetector._build_knowledge_graph").get("auto_param_L845_33", 0.25), get_parameter_loade
r().get("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._build_knowl
edge_graph").get("auto_param_L845_39", 0.2), get_parameter_loader().get("saaaaaa.analysis.
contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_par
am_L845_44", 0.15), get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.P
olicyContradictionDetector._build_knowledge_graph").get("auto_param_L845_50", 0.1)])
        scores = np.array([
            1 - contradiction_density,
            semantic_coherence,
            temporal_consistency,
            objective_alignment,
            1 - graph_fragmentation
        ])

        # Harmonic mean ponderada para penalizar valores bajos
        coherence_score = np.sum(weights) / np.sum(weights / np.maximum(scores, get_parame
ter_loader().get("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._bu
ild_knowledge_graph").get("auto_param_L855_80", 0.01)))

        # Entropía de contradicciones
        contradiction_entropy = self._calculate_contradiction_entropy(contradictions)

        # Complejidad sintáctica del documento
        syntactic_complexity = self._calculate_syntactic_complexity(text)

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

    def _calculate_global_semantic_coherence(
            self,
            statements: list[PolicyStatement]
    ) -> float:
        """Calcula coherencia semántica global usando embeddings"""
        if len(statements) < 2:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._build_knowledge_graph").get("auto_param_L881_19", 1.0)

        # Calcular matriz de similitud
```

```
        embeddings = [s.embedding for s in statements if s.embedding is not None]
        if len(embeddings) < 2:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._build_knowledge_graph").get("auto_param_L886_19", 0.5)

        similarity_matrix = cosine_similarity(embeddings)

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

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

        coherence = mean_similarity * (1 - min(get_parameter_loader().get("saaaaaa.analysi
s.contradiction_deteccion.PolicyContradictionDetector._build_knowledge_graph").get("auto_p
aram_L899_47", 0.5), std_similarity))

        return float(coherence)

    def _calculate_objective_alignment(
            self,
            statements: list[PolicyStatement]
    ) -> float:
        """Calcula alineación entre objetivos declarados"""
        objective_statements = [
            s for s in statements
            if s.semantic_role in ['objective', 'goal', 'target']
        ]

        if len(objective_statements) < 2:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._build_knowledge_graph").get("auto_param_L914_19", 1.0)

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

        if alignment_scores:
            return float(np.mean(alignment_scores))
        return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Policy
ContradictionDetector._build_knowledge_graph").get("auto_param_L927_15", 0.5)

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._calculate_graph_fragmentation")
    def _calculate_graph_fragmentation(self) -> float:
        """Calcula fragmentación del grafo de conocimiento"""
        if self.knowledge_graph.number_of_nodes() == 0:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._calculate_graph_fragmentation").get("auto_param_L933_19", 0.0)

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

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

        return float(fragmentation)

    def _calculate_contradiction_entropy(
```

```
        self,
        contradictions: list[ContradictionEvidence]
    ) -> float:
        """Calcula entropía de distribución de tipos de contradicción"""
        if not contradictions:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._calculate_graph_fragmentation").get("auto_param_L950_19", 0.0)

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

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

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

        # Normalizar por entropía máxima
        max_entropy = np.log2(len(ContradictionType))
        normalized_entropy = entropy / max_entropy if max_entropy > 0 else 0

        return float(normalized_entropy)

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._calculate_syntactic_complexity")
    def _calculate_syntactic_complexity(self, text: str) -> float:
        """Calcula complejidad sintáctica del documento"""
        doc = self.nlp(text[:5000])  # Limitar para eficiencia

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

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

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

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

        # Combinar métricas
        complexity = (
            min(get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.P
olicyContradictionDetector._calculate_syntactic_complexity").get("auto_param_L993_20",
1.0), avg_sentence_length / 50) * get_parameter_loader().get("saaaaaa.analysis.contradicti
on_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity").get("auto_param
_L993_53", 0.3) +
            min(get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.P
olicyContradictionDetector._calculate_syntactic_complexity").get("auto_param_L994_20",
1.0), avg_dependency_depth / 10) * get_parameter_loader().get("saaaaaa.analysis.contradict
ion_deteccion.PolicyContradictionDetector._calculate_syntactic_complexity").get("auto_para
m_L994_54", 0.3) +
            ttr * get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion
.PolicyContradictionDetector._calculate_syntactic_complexity").get("auto_param_L995_22",
0.4)
        )

        return float(complexity)
```

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

    def _calculate_confidence_interval(
            self,
            score: float,
            n_observations: int
    ) -> tuple[float, float]:
        """Calcula intervalo de confianza del 95% para el score"""
        # Usar distribución t de Student para muestras pequeñas
        if n_observations < 30:
            # Error estándar estimado
            se = np.sqrt(score * (1 - score) / n_observations)
            # Valor crítico t para 95% de confianza
            t_critical = stats.t.ppf(get_parameter_loader().get("saaaaaa.analysis.contradi
ction_deteccion.PolicyContradictionDetector._get_dependency_depth").get("auto_param_L1021_
37", 0.975), n_observations - 1)
            margin = t_critical * se
        else:
            # Usar distribución normal para muestras grandes
            se = np.sqrt(score * (1 - score) / n_observations)
            margin = 1.96 * se

        return (
            max(get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Polic
yContradictionDetector._get_dependency_depth").get("auto_param_L1029_16", 0.0), score -
margin),
            min(get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Polic
yContradictionDetector._get_dependency_depth").get("auto_param_L1030_16", 1.0), score +
margin)
        )

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

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

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

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

            elif cont_type == ContradictionType.RESOURCE_ALLOCATION_MISMATCH:
                recommendations.append({
                    "type": "budget_reallocation",
                    "priority": "critical",
                    "description": "Revisar asignación presupuestal",
                    "specific_actions": [
                        "Realizar análisis de suficiencia presupuestal",
                        "Priorizar programas según impacto",
                        "Identificar fuentes alternativas de financiación"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

            elif cont_type == ContradictionType.SEMANTIC_OPPOSITION:
                recommendations.append({
                    "type": "conceptual_clarification",
                    "priority": "medium",
                    "description": "Clarificar conceptos y objetivos opuestos",
                    "specific_actions": [
                        "Realizar sesiones de alineación estratégica",
                        "Definir glosario de términos unificado",
                        "Establecer jerarquía clara de objetivos"
                    ],
                    "affected_sections": self._identify_affected_sections(conflicts)
                })

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

        return recommendations

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

        return list(affected)

    def _serialize_contradiction(
            self,
            contradiction: ContradictionEvidence
    ) -> dict[str, Any]:
        """Serializa evidencia de contradicción para output"""
```

```
        return {
            "statement_1": contradiction.statement_a.text,
            "statement_2": contradiction.statement_b.text,
            "position_1": contradiction.statement_a.position,
            "position_2": contradiction.statement_b.position,
            "contradiction_type": contradiction.contradiction_type.name,
            "confidence": float(contradiction.confidence),
            "severity": float(contradiction.severity),
            "semantic_similarity": float(contradiction.semantic_similarity),
            "logical_conflict_score": float(contradiction.logical_conflict_score),
            "temporal_consistency": contradiction.temporal_consistency,
            "numerical_divergence": float(
                contradiction.numerical_divergence) if contradiction.numerical_divergence
else None,
            "statistical_significance": float(
                contradiction.statistical_significance) if
contradiction.statistical_significance else None,
            "affected_dimensions": [d.value for d in contradiction.affected_dimensions],
            "resolution_suggestions": contradiction.resolution_suggestions,
            "graph_path": contradiction.graph_path
        }

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

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

    # Métodos auxiliares

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

        # Patrones de fechas
        date_patterns = [
            r'\d{1,2}\s+de\s+\w+\s+de\s+\d{4}',
            r'\d{4}-\d{2}-\d{2}',
            r'(enero|febrero|marzo|abril|mayo|junio|julio|agosto|septiembre|octubre|noviem
bre|diciembre)\s+\d{4}',
            r'(Q[1-4]|trimestre\s+[1-4])\s+\d{4}',
            r'20\d{2}',
            r'(corto|mediano|largo)\s+plazo',
            r'(primer|segundo|tercer|cuarto)\s+(año|semestre|trimestre)'
        ]

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

        return markers

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
```

```
or._extract_quantitative_claims")
    def _extract_quantitative_claims(self, text: str) -> list[dict[str, Any]]:
        """Extrae afirmaciones cuantitativas estructuradas"""
        claims = []

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

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

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

        return claims

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._parse_number")
    def _parse_number(self, text: str) -> float:
        """Parsea número desde texto"""
        try:
            # Reemplazar coma decimal
            normalized = text.replace(',', '.')
            return float(normalized)
        except ValueError:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._parse_number").get("auto_param_L1227_19", 0.0)

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

        # Patrones de recursos específicos de PDM colombiano
        resource_patterns = [
            (r'SGP\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'SGP'),
            (r'regalías\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?', 'regalías'),
            (r'recursos\s+propios\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?',
'recursos_propios'),
            (r'cofinanciación\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?',
'cofinanciación'),
            (r'presupuesto\s+total\s*[:\s]*\$?\s*(\d+(?:[.,]\d+)?)\s*(millones?)?',
'presupuesto_total')
        ]

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

```
                amount = amount * 1000000 if amount else None
            resources.append((resource_type, amount))

        return resources

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._determine_semantic_role")
    def _determine_semantic_role(self, sent) -> str | None:
        """Determina el rol semántico de una oración"""
        # Safely extract text (handles both strings and spacy objects)
        text_lower = safe_text_extract(sent).lower()

        role_patterns = {
            'objective': ['objetivo', 'meta', 'propósito', 'finalidad'],
            'strategy': ['estrategia', 'línea', 'eje', 'pilar'],
            'action': ['implementar', 'ejecutar', 'desarrollar', 'realizar'],
            'indicator': ['indicador', 'medir', 'evaluar', 'monitorear'],
            'resource': ['presupuesto', 'recurso', 'financiación', 'inversión'],
            'constraint': ['limitación', 'restricción', 'condición', 'requisito']
        }

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

        return None

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

        # Buscar referencias a otras secciones
        reference_patterns = [
            r'como\s+se\s+menciona\s+en',
            r'según\s+lo\s+establecido\s+en',
            r'de\s+acuerdo\s+con',
            r'en\s+línea\s+con',
            r'siguiendo\s+lo\s+dispuesto'
        ]

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

        return dependencies

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

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._calculate_similarity")
    def _calculate_similarity(self, stmt_a: PolicyStatement, stmt_b: PolicyStatement) ->
float:
        """Calcula similaridad entre dos declaraciones"""
        if stmt_a.embedding is not None and stmt_b.embedding is not None:
            return float(1 - cosine(stmt_a.embedding, stmt_b.embedding))
```

```
        return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Policy
ContradictionDetector._calculate_similarity").get("auto_param_L1310_15", 0.0)

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._classify_contradiction")
    def _classify_contradiction(self, text: str) -> float:
        """Clasifica probabilidad de contradicción en texto"""
        try:
            result = self.contradiction_classifier(text)
            # Buscar score de contradicción
            for item in result:
                if 'contradiction' in item['label'].lower():
                    return item['score']
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._classify_contradiction").get("auto_param_L1321_19", 0.0)
        except Exception as e:
            logger.warning(f"Error en clasificación de contradicción: {e}")
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._classify_contradiction").get("auto_param_L1324_19", 0.0)

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._get_domain_weight")
    def _get_domain_weight(self, dimension: PolicyDimension) -> float:
        """Obtiene peso específico del dominio"""
        weights = {
            PolicyDimension.DIAGNOSTICO: get_parameter_loader().get("saaaaaa.analysis.cont
radiction_deteccion.PolicyContradictionDetector._get_domain_weight").get("auto_param_L1330
_41", 0.8),
            PolicyDimension.ESTRATEGICO: 1.2,
            PolicyDimension.PROGRAMATICO: get_parameter_loader().get("saaaaaa.analysis.con
tradiction_deteccion.PolicyContradictionDetector._get_domain_weight").get("auto_param_L133
2_42", 1.0),
            PolicyDimension.FINANCIERO: 1.5,
            PolicyDimension.SEGUIMIENTO: get_parameter_loader().get("saaaaaa.analysis.cont
radiction_deteccion.PolicyContradictionDetector._get_domain_weight").get("auto_param_L1334
_41", 0.9),
            PolicyDimension.TERRITORIAL: 1.1
        }
        return weights.get(dimension, get_parameter_loader().get("saaaaaa.analysis.contrad
iction_deteccion.PolicyContradictionDetector._get_domain_weight").get("auto_param_L1337_38
", 1.0))

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._suggest_resolutions")
    def _suggest_resolutions(self, contradiction_type: ContradictionType) -> list[str]:
        """Sugiere resoluciones específicas por tipo de contradicción"""
        suggestions = {
            ContradictionType.NUMERICAL_INCONSISTENCY: [
                "Verificar fuentes de datos y metodologías de cálculo",
                "Establecer línea base única con validación técnica",
                "Documentar supuestos y proyecciones utilizadas"
            ],
            ContradictionType.TEMPORAL_CONFLICT: [
                "Revisar cronograma maestro del plan",
                "Validar secuencia lógica de actividades",
                "Ajustar plazos según capacidad institucional"
            ],
            ContradictionType.SEMANTIC_OPPOSITION: [
                "Realizar taller de alineación conceptual",
                "Clarificar definiciones en glosario técnico",
                "Priorizar objetivos según Plan Nacional de Desarrollo"
            ],
            ContradictionType.RESOURCE_ALLOCATION_MISMATCH: [
                "Realizar análisis de brechas financieras",
                "Priorizar inversiones según impacto social",
                "Explorar fuentes alternativas de financiación"
            ],
            ContradictionType.LOGICAL_INCOMPATIBILITY: [
```

```
                "Revisar cadena de valor de programas",
                "Validar teoría de cambio del plan",
                "Eliminar duplicidades y solapamientos"
            ]
        }
        return suggestions.get(contradiction_type, ["Revisar y ajustar según contexto"])

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

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

        return context_similarity > get_parameter_loader().get("saaaaaa.analysis.contradic
tion_deteccion.PolicyContradictionDetector._are_comparable_claims").get("auto_param_L1384_
36", 0.6)

    @calibrated_method("saaaaaa.analysis.contradiction_deteccion.PolicyContradictionDetect
or._text_similarity")
    def _text_similarity(self, text_a: str, text_b: str) -> float:
        """Calcula similaridad simple entre textos"""
        if not text_a or not text_b:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._text_similarity").get("auto_param_L1390_19", 0.0)

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

        if not tokens_a or not tokens_b:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._text_similarity").get("auto_param_L1397_19", 0.0)

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

        return len(intersection) / len(union) if union else get_parameter_loader().get("sa
aaaaa.analysis.contradiction_deteccion.PolicyContradictionDetector._text_similarity").get(
"auto_param_L1403_60", 0.0)

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

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

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

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

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

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

        if pooled_value == 0:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._text_similarity").get("auto_param_L1440_19", 1.0)  # No
significativo

        # Estimación conservadora de error estándar
        se = pooled_value * get_parameter_loader().get("saaaaaa.analysis.contradiction_det
eccion.PolicyContradictionDetector._text_similarity").get("auto_param_L1443_28", 0.1)  #
10% de error estimado

        if se == 0:
            return get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._text_similarity").get("auto_param_L1446_19", 0.0)  # Altamente
significativo

        # Estadístico t
        t_stat = diff / se

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

        return p_value

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

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

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

        # Si una tiene negación y otra no, y son similares, hay conflicto
        if has_negation_a != has_negation_b:
            similarity = self._calculate_similarity(stmt_a, stmt_b)
            if similarity > get_parameter_loader().get("saaaaaa.analysis.contradiction_det
```

```
eccion.PolicyContradictionDetector._has_logical_conflict").get("auto_param_L1481_28",
0.7):
                return True

        return False

    def _are_conflicting_allocations(
            self,
            amount_a: float,
            amount_b: float,
            total: float
    ) -> bool:
        """Determina si las asignaciones de recursos están en conflicto"""
        # Si la suma excede el total disponible
        if amount_a + amount_b > total * 1.1:  # 10% de margen
            return True

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

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

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

        # Por defecto, relación de similaridad
        return "related"

    def _calculate_severity(
            self,
            stmt_a: PolicyStatement,
            stmt_b: PolicyStatement
    ) -> float:
        """Calcula la severidad de una contradicción entre declaraciones"""
        severity = get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._has_logical_conflict").get("severity", 0.5) # Refactored

        # Incrementar si las declaraciones están en la misma dimensión
        if stmt_a.dimension == stmt_b.dimension:
            severity += get_parameter_loader().get("saaaaaa.analysis.contradiction_detecci
on.PolicyContradictionDetector._has_logical_conflict").get("auto_param_L1532_24", 0.2)

        # Incrementar si tienen muchas entidades en común
        common_entities = set(stmt_a.entities) & set(stmt_b.entities)
        if len(common_entities) > 0:
            severity += min(get_parameter_loader().get("saaaaaa.analysis.contradiction_det
eccion.PolicyContradictionDetector._has_logical_conflict").get("auto_param_L1537_28",
0.2), len(common_entities) * get_parameter_loader().get("saaaaaa.analysis.contradiction_de
teccion.PolicyContradictionDetector._has_logical_conflict").get("auto_param_L1537_56",
0.05))
```

```
        # Incrementar si tienen marcadores temporales en conflicto
        if stmt_a.temporal_markers and stmt_b.temporal_markers:
            severity += get_parameter_loader().get("saaaaaa.analysis.contradiction_detecci
on.PolicyContradictionDetector._has_logical_conflict").get("auto_param_L1541_24", 0.1)

        return min(get_parameter_loader().get("saaaaaa.analysis.contradiction_deteccion.Po
licyContradictionDetector._has_logical_conflict").get("auto_param_L1543_19", 1.0),
severity)

===== FILE: src/saaaaaa/analysis/derek_beach.py =====
#!/usr/bin/env python3
"""
Causal Deconstruction and Audit Framework (CDAF) v2.0
Framework de Producción para Análisis Causal de Planes de Desarrollo Territorial

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

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

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

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

if TYPE_CHECKING:
    import fitz

# Core dependencies
try:
    import networkx as nx
    import numpy as np
    import pandas as pd
    import spacy
    import yaml
    from fuzzywuzzy import fuzz, process
    from pydantic import BaseModel, Field, ValidationError, validator
    from pydot import Dot, Edge, Node
    from scipy import stats
    from scipy.spatial.distance import cosine
    from scipy.special import rel_entr
except ImportError as e:
    print(f"ERROR: Dependencia faltante. Ejecute: pip install {e.name}")
    sys.exit(1)
```

```
# DNP Standards Integration
try:
    from dnp_integration import ValidadorDNP, validar_plan_desarrollo_completo

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

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

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

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

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

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

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

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

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

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

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

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

    STRAW-IN-WIND [N: Low, S: Low]:
    - Pass/Fail → Marginal confidence change
    - Used for preliminary screening
```

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

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

    Beach calibration:
    - Necessity > 0.7 → High necessity
    - Sufficiency > 0.7 → High sufficiency
    """
    high_n = necessity > 0.7
    high_s = sufficiency > 0.7

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

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

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

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

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

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

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

```
            posterior = max(0.05, prior / min(bayes_factor, 2.0))
            return posterior, "STRAW_IN_WIND: Weak disconfirmation"

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

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

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

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFException._format_message")
    def _format_message(self) -> str:
        """Format error message with structured information"""
        parts = ["[CDAF Error]"]
        if self.stage:
            parts.append(f"[Stage: {self.stage}]")
        parts.append(self.message)
        if self.details:
            parts.append(f"Details: {json.dumps(self.details, indent=2)}")
        return " ".join(parts)

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

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

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

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

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

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

class BayesianThresholdsConfig(BaseModel):
    """Bayesian inference thresholds configuration"""
    kl_divergence: float = Field(
        default=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_
dict").get("auto_param_L255_16", 0.01),
        ge=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L256_11", 0.0),
        le=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
```

```
).get("auto_param_L257_11", 1.0),
        description="KL divergence threshold for convergence"
    )
    convergence_min_evidence: int = Field(
        default=2,
        ge=1,
        description="Minimum evidence count for convergence check"
    )
    prior_alpha: float = Field(
        default=2.0,
        ge=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L267_11", 0.1),
        description="Default alpha parameter for Beta prior"
    )
    prior_beta: float = Field(
        default=2.0,
        ge=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L272_11", 0.1),
        description="Default beta parameter for Beta prior"
    )
    laplace_smoothing: float = Field(
        default=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_
dict").get("auto_param_L276_16", 1.0),
        ge=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L277_11", 0.0),
        description="Laplace smoothing parameter"
    )

class MechanismTypeConfig(BaseModel):
    """Mechanism type prior probabilities"""
    administrativo: float = Field(default=get_parameter_loader().get("saaaaaa.analysis.der
ek_beach.CDAFException.to_dict").get("auto_param_L283_42", 0.30), ge=get_parameter_loader(
).get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L283_51",
0.0), le=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").
get("auto_param_L283_59", 1.0))
    tecnico: float = Field(default=get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.CDAFException.to_dict").get("auto_param_L284_35", 0.25), ge=get_parameter_loader().get("
saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L284_44", 0.0), le=ge
t_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_p
aram_L284_52", 1.0))
    financiero: float = Field(default=get_parameter_loader().get("saaaaaa.analysis.derek_b
each.CDAFException.to_dict").get("auto_param_L285_38", 0.20), ge=get_parameter_loader().ge
t("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L285_47", 0.0), le
=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("aut
o_param_L285_55", 1.0))
    politico: float = Field(default=get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.CDAFException.to_dict").get("auto_param_L286_36", 0.15), ge=get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L286_45", 0.0), le=g
et_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_
param_L286_53", 1.0))
    mixto: float = Field(default=get_parameter_loader().get("saaaaaa.analysis.derek_beach.
CDAFException.to_dict").get("auto_param_L287_33", 0.10), ge=get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L287_42", 0.0), le=get_
parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_par
am_L287_50", 1.0))

    @validator('*', pre=True, always=True)
    def check_sum_to_one(cls, v, values):
        """Validate that probabilities sum to approximately get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L291_60", 1.0)"""
        if len(values) == 4:  # All fields loaded
            total = sum(values.values()) + v
            if abs(total - get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFEx
ception.to_dict").get("auto_param_L294_27", 1.0)) > get_parameter_loader().get("saaaaaa.an
alysis.derek_beach.CDAFException.to_dict").get("auto_param_L294_34", 0.01):
                raise ValueError(f"Mechanism type priors must sum to get_parameter_loader(
).get("saaaaaa.analysis.derek_beach.CDAFException.to_dict").get("auto_param_L295_69",
1.0), got {total}")
```

```
        return v

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

class SelfReflectionConfig(BaseModel):
    """Self-reflective learning configuration"""
    enable_prior_learning: bool = Field(
        default=False,
        description="Enable learning from audit feedback to update priors"
    )
    feedback_weight: float = Field(
        default=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_
dict").get("auto_param_L325_16", 0.1),
        ge=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L326_11", 0.0),
        le=get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFException.to_dict"
).get("auto_param_L327_11", 1.0),
        description="Weight for feedback in prior updates (0=ignore, 1=full)"
    )
    prior_history_path: str | None = Field(
        default=None,
        description="Path to save/load historical priors"
    )
    min_documents_for_learning: int = Field(
        default=5,
        ge=1,
        description="Minimum documents before applying learned priors"
    )

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

```
    )
    performance: PerformanceConfig = Field(
        default_factory=PerformanceConfig,
        description="Performance and optimization settings"
    )
    self_reflection: SelfReflectionConfig = Field(
        default_factory=SelfReflectionConfig,
        description="Self-reflective learning configuration"
    )

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

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

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

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

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

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

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

@dataclass
class MetaNode:
    """Comprehensive node structure for goals/metas"""
    id: str
    text: str
    type: NodeType
    baseline: float | str | None = None
    target: float | str | None = None
    unit: str | None = None
    responsible_entity: str | None = None
    entity_activity: EntityActivity | None = None
    financial_allocation: float | None = None
    unit_cost: float | None = None
    rigor_status: RigorStatus = "sin_evaluar"
```

```
    dynamics: DynamicsType = "indefinido"
    test_type: TestType = "straw_in_wind"
    contextual_risks: list[str] = field(default_factory=list)
    causal_justification: list[str] = field(default_factory=list)
    audit_flags: list[str] = field(default_factory=list)
    confidence_score: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDA
FException.to_dict").get("auto_param_L434_30", 0.0)

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

    def __init__(self, config_path: Path) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config_path = config_path
        self.config: dict[str, Any] = {}
        self.validated_config: CDAFConfigSchema | None = None
        # HARMONIC FRONT 4: Track uncertainty over iterations
        self._uncertainty_history: list[float] = []
        self._load_config()
        self._validate_config()
        self._load_uncertainty_history()

    @calibrated_method("saaaaaa.analysis.derek_beach.ConfigLoader._load_config")
    def _load_config(self) -> None:
        """Load YAML configuration file"""
        # Delegate to factory for I/O operation
        from .factory import load_yaml

        try:
            self.config = load_yaml(self.config_path)
            self.logger.info(f"Configuración cargada desde {self.config_path}")
        except FileNotFoundError:
            self.logger.warning(f"Archivo de configuración no encontrado:
{self.config_path}")
            self._load_default_config()
        except Exception as e:
            raise CDAFConfigError(
                "Error cargando configuración",
                details={'path': str(self.config_path), 'error': str(e)},
                stage="config_load",
                recoverable=True
            )

    @calibrated_method("saaaaaa.analysis.derek_beach.ConfigLoader._load_default_config")
    def _load_default_config(self) -> None:
        """Load default configuration if custom fails"""
        self.config = {
            'patterns': {
                'section_titles': r'^(?:CAPÍTULO|ARTÍCULO|PARTE)\s+[\dIVX]+',
                'goal_codes': r'[MP][RIP]-\d{3}',
                'numeric_formats': r'[\d,]+(?:\.\d+)?%?',
                'table_headers': r'(?:PROGRAMA|META|INDICADOR|LÍNEA BASE|VALOR ESPERADO)',
                'financial_headers': r'(?:PRESUPUESTO|VALOR|MONTO|INVERSIÓN)'
            },
            'lexicons': {
                'causal_logic': [
                    'gracias a', 'con el fin de', 'para lograr', 'mediante',
                    'a través de', 'como resultado de', 'debido a', 'porque',
                    'por medio de', 'permitirá', 'contribuirá a'
                ],
                'goal_classification': {
                    'tasa': 'decreciente',
                    'índice': 'constante',
                    'número': 'suma',
                    'porcentaje': 'constante',
                    'cantidad': 'suma',
                    'cobertura': 'suma'
                },
```

```
            'contextual_factors': [
                'riesgo', 'amenaza', 'obstáculo', 'limitación',
                'restricción', 'desafío', 'brecha', 'déficit',
                'vulnerabilidad', 'hipótesis alternativa'
            ],
            'administrative_keywords': [
                'gestión', 'administración', 'coordinación', 'regulación',
                'normativa', 'institucional', 'gobernanza', 'reglamento',
                'decreto', 'resolución', 'acuerdo'
            ]
        },
        'entity_aliases': {
            'SEC GOB': 'Secretaría de Gobierno',
            'SEC PLAN': 'Secretaría de Planeación',
            'SEC HAC': 'Secretaría de Hacienda',
            'SEC SALUD': 'Secretaría de Salud',
            'SEC EDU': 'Secretaría de Educación',
            'SEC INFRA': 'Secretaría de Infraestructura'
        },
        'verb_sequences': {
            'diagnosticar': 1,
            'identificar': 2,
            'analizar': 3,
            'diseñar': 4,
            'planificar': 5,
            'implementar': 6,
            'ejecutar': 7,
            'monitorear': 8,
            'evaluar': 9
        },
        # Bayesian thresholds - now externalized
        'bayesian_thresholds': {
            'kl_divergence': get_parameter_loader().get("saaaaaa.analysis.derek_beach.
ConfigLoader._load_default_config").get("auto_param_L527_33", 0.01),
            'convergence_min_evidence': 2,
            'prior_alpha': 2.0,
            'prior_beta': 2.0,
            'laplace_smoothing': get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.ConfigLoader._load_default_config").get("auto_param_L531_37", 1.0)
        },
        # Mechanism type priors - now externalized
        'mechanism_type_priors': {
            'administrativo': get_parameter_loader().get("saaaaaa.analysis.derek_beach
.ConfigLoader._load_default_config").get("auto_param_L535_34", 0.30),
            'tecnico': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Config
Loader._load_default_config").get("auto_param_L536_27", 0.25),
            'financiero': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Con
figLoader._load_default_config").get("auto_param_L537_30", 0.20),
            'politico': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Confi
gLoader._load_default_config").get("auto_param_L538_28", 0.15),
            'mixto': get_parameter_loader().get("saaaaaa.analysis.derek_beach.ConfigLo
ader._load_default_config").get("auto_param_L539_25", 0.10)
        },
        # Performance settings
        'performance': {
            'enable_vectorized_ops': True,
            'enable_async_processing': False,
            'max_context_length': 1000,
            'cache_embeddings': True
        },
        # Self-reflection settings
        'self_reflection': {
            'enable_prior_learning': False,
            'feedback_weight': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.ConfigLoader._load_default_config").get("auto_param_L551_35", 0.1),
            'prior_history_path': None,
            'min_documents_for_learning': 5
        }
```

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

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

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

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

    @calibrated_method("saaaaaa.analysis.derek_beach.ConfigLoader.get_bayesian_threshold")
    def get_bayesian_threshold(self, key: str) -> float:
        """Get Bayesian threshold with type safety"""
        if self.validated_config:
            return getattr(self.validated_config.bayesian_thresholds, key)
        return self.get(f'bayesian_thresholds.{key}', get_parameter_loader().get("saaaaaa.
analysis.derek_beach.ConfigLoader.get_bayesian_threshold").get("auto_param_L607_54",
0.01))

    @calibrated_method("saaaaaa.analysis.derek_beach.ConfigLoader.get_mechanism_prior")
    def get_mechanism_prior(self, mechanism_type: str) -> float:
        """Get mechanism type prior probability with type safety"""
        if self.validated_config:
            return getattr(self.validated_config.mechanism_type_priors, mechanism_type, ge
t_parameter_loader().get("saaaaaa.analysis.derek_beach.ConfigLoader.get_mechanism_prior").
get("auto_param_L613_88", 0.0))
        return self.get(f'mechanism_type_priors.{mechanism_type}', get_parameter_loader().
get("saaaaaa.analysis.derek_beach.ConfigLoader.get_mechanism_prior").get("auto_param_L614_
67", 0.0))
```

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

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

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

        feedback_weight = self.validated_config.self_reflection.feedback_weight

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

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

        # NEW: Apply penalty factors for failing mechanism types
        if 'penalty_factors' in feedback_data:
            penalty_weight = feedback_weight * 1.5  # Heavier penalty than positive
feedback
            for mech_type, penalty_factor in feedback_data['penalty_factors'].items():
                if hasattr(self.validated_config.mechanism_type_priors, mech_type):
                    current_prior = getattr(self.validated_config.mechanism_type_priors,
mech_type)
                    # Apply penalty: reduce prior for frequently failing types
                    penalized_prior = current_prior * penalty_factor
                    # Blend with current
                    updated_prior = (1 - penalty_weight) * current_prior + penalty_weight
* penalized_prior
                    setattr(self.validated_config.mechanism_type_priors, mech_type,
updated_prior)
                    self.config['mechanism_type_priors'][mech_type] = updated_prior
                    self.logger.info(f"Applied penalty to {mech_type}: {current_prior:.4f}
 -> {updated_prior:.4f}")

        # NEW: Heavy penalty for "miracle" mechanisms failing necessity/sufficiency
        test_failures = feedback_data.get('test_failures', {})
```

```
        if test_failures.get('necessity_failures', 0) > 0 or
test_failures.get('sufficiency_failures', 0) > 0:
            # If failures exist, apply additional penalty to 'politico' (often "miracle"
type)
            # and 'mixto' (vague mechanism types)
            miracle_types = ['politico', 'mixto']
            miracle_penalty = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Con
figLoader.update_priors_from_feedback").get("miracle_penalty", 0.85) # Refactored
            for mech_type in miracle_types:
                if hasattr(self.validated_config.mechanism_type_priors, mech_type):
                    current_prior = getattr(self.validated_config.mechanism_type_priors,
mech_type)
                    updated_prior = current_prior * miracle_penalty
                    setattr(self.validated_config.mechanism_type_priors, mech_type,
updated_prior)
                    self.config['mechanism_type_priors'][mech_type] = updated_prior
                    self.logger.info(
                        f"Miracle mechanism penalty for {mech_type}: {current_prior:.4f}
-> {updated_prior:.4f}")

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

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

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

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

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

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

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

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

        HARMONIC FRONT 4 ENHANCEMENT:
        - Tracks uncertainty reduction over iterations
```

```
        - Records penalty applications and test failures
        """
        if not self.validated_config or not
self.validated_config.self_reflection.prior_history_path:
            return

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

            # Load existing history if available
            # Delegate to factory for I/O operation
            from .factory import load_json, save_json

            history_records = []
            if history_path.exists():
                try:
                    existing_data = load_json(history_path)
                    if isinstance(existing_data, list):
                        history_records = existing_data
                    elif isinstance(existing_data, dict) and 'history' in existing_data:
                        history_records = existing_data['history']
                except json.JSONDecodeError:
                    self.logger.warning("Existing history file corrupted, starting fresh")

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

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

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

            history_records.append(history_record)

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

            save_json(history_data, history_path)

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


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

        HARMONIC FRONT 4: Required for tracking ≥5% reduction over 10 iterations
```

```
        """
        if not self.validated_config or not
self.validated_config.self_reflection.prior_history_path:
            return

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

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

    @calibrated_method("saaaaaa.analysis.derek_beach.ConfigLoader.check_uncertainty_reduct
ion_criterion")
    def check_uncertainty_reduction_criterion(self, current_uncertainty: float) ->
dict[str, Any]:
        """
        Check if mean mechanism_type uncertainty has decreased ≥5% over 10 iterations

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

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

        result = {
            'current_uncertainty': current_uncertainty,
            'iterations_tracked': len(recent_history),
            'criterion_met': False,
            'reduction_percent': get_parameter_loader().get("saaaaaa.analysis.derek_beach.
ConfigLoader.check_uncertainty_reduction_criterion").get("auto_param_L830_33", 0.0),
            'status': 'insufficient_data'
        }

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

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

                self.logger.info(
                    f"Uncertainty reduction over 10 iterations: {reduction_percent:.2f}% "
                    f"(criterion: ≥5%, met: {result['criterion_met']})"
                )
        else:
            self.logger.info(
                f"Uncertainty tracking: {len(recent_history)}/10 iterations "
```

```
                f"(need {10 - len(recent_history)} more for criterion check)"
            )

        return result

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

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

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

                @self.retry_handler.with_retry(
                    DependencyType.PDF_PARSER,
                    operation_name="open_pdf",
                    exceptions=(IOError, OSError, RuntimeError)
                )
                def load_with_retry():
                    # Delegate to factory for I/O operation
                    from .factory import open_pdf_with_fitz

                    doc = open_pdf_with_fitz(pdf_path)
                    self.logger.info(f"PDF cargado: {pdf_path.name} ({len(doc)} páginas)")
                    return doc

                self.document = load_with_retry()
                self.metadata = self.document.metadata
                return True
            except Exception as e:
                self.logger.error(f"Error cargando PDF: {e}")
                return False
        else:
            # Fallback without retry
            # Delegate to factory for I/O operation
            from .factory import open_pdf_with_fitz

            try:
                self.document = open_pdf_with_fitz(pdf_path)
                self.metadata = self.document.metadata
                self.logger.info(f"PDF cargado: {pdf_path.name} ({len(self.document)}
páginas)")
                return True
            except Exception as e:
                self.logger.error(f"Error cargando PDF: {e}")
                return False

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

        text_parts = []
        for page_num, page in enumerate(self.document, 1):
            try:
                text = page.get_text()
```

```
                text_parts.append(text)
                self.logger.debug(f"Texto extraído de página {page_num}")
            except Exception as e:
                self.logger.warning(f"Error extrayendo texto de página {page_num}: {e}")

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

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

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

        for page_num, page in enumerate(self.document, 1):
            try:
                tabs = page.find_tables()
                if tabs:
                    for tab in tabs:
                        try:
                            df = pd.DataFrame(tab.extract())
                            if not df.empty and len(df.columns) > 1:
                                # Check if this is a relevant table
                                header_text = ' '.join(str(cell) for cell in df.iloc[0] if
 cell)
                                if table_pattern.search(header_text):
                                    self.tables.append(df)
                                    self.logger.info(f"Tabla extraída de página
{page_num}: {df.shape}")
                        except Exception as e:
                            self.logger.warning(f"Error procesando tabla en página
{page_num}: {e}")
            except Exception as e:
                self.logger.debug(f"Error extrayendo tablas de página {page_num}: {e}")

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

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

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

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

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

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

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

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

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

        # Extract causal connections
        self._extract_causal_links(text)

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

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

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

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

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

        self.logger.info(f"Metas extraídas: {len(goals)}")
        return goals

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

        # Extract numerical values
```

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

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

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

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

    return goal

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

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

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

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

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

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

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

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

        return context.strip()

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


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

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

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

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

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

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

                    # Calculate evidence components
                    evidence = {
                        'keyword': keyword,
                        'logic': logic,
                        'match_position': match.start(),
                        'semantic_distance': self._calculate_semantic_distance(source,
target),
                        'type_transition_prior':
self._calculate_type_transition_prior(source, target),
                        'language_specificity':
self._calculate_language_specificity(keyword, None, match_context),
                        'temporal_coherence': self._assess_temporal_coherence(source,
target),
                        'financial_consistency':
self._assess_financial_consistency(source, target),
```

```
                    'textual_proximity': self._calculate_textual_proximity(source,
target, text)
                }

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

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

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

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

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

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

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

            # AUDIT POINT 2.1: Structural Veto (D6-Q2)
            # TeoriaCambio validation - caps Bayesian posterior ≤get_parameter_loader().ge
t("saaaaaa.analysis.derek_beach.CausalExtractor._extract_causal_links").get("auto_param_L1
217_65", 0.6) for impermissible links
            # Implements axiomatic-Bayesian fusion per Goertz & Mahoney 2012
            structural_violation = self._check_structural_violation(source, target)
            if structural_violation:
                # Deterministic veto: cap posterior at get_parameter_loader().get("saaaaaa
.analysis.derek_beach.CausalExtractor._extract_causal_links").get("auto_param_L1221_55",
0.6) despite high semantic evidence
                original_posterior = posterior_mean
                posterior_mean = min(posterior_mean, get_parameter_loader().get("saaaaaa.a
nalysis.derek_beach.CausalExtractor._extract_causal_links").get("auto_param_L1223_53",
0.6))
                self.logger.warning(
                    f"STRUCTURAL VETO (D6-Q2): Link {source}→{target} violates causal
hierarchy. "
                    f"Posterior capped from {original_posterior:.3f} to
{posterior_mean:.3f}. "
                    f"Violation: {structural_violation}"
                )

            # Check convergence (require minimum evidence count)
            converged = (len(kl_divs) >= convergence_min_evidence and
                         len(kl_divs) > 0 and kl_divs[-1] < kl_threshold)
```

```
            final_kl = kl_divs[-1] if len(kl_divs) > 0 else get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.CausalExtractor._extract_causal_links").get("auto_param_L1233_6
0", 0.0)

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

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

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

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

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

        Current implementation prioritizes determinism over speed.
        Enable performance.cache_embeddings in config for production use.
        """
        try:
            source_node = self.nodes.get(source)
            target_node = self.nodes.get(target)

            if not source_node or not target_node:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._calculate_semantic_distance").get("auto_param_L1286_23", 0.5)

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

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

```
            if source_doc.vector.any() and target_doc.vector.any():
                # Calculate cosine similarity (1 - distance)
                # PERFORMANCE NOTE: Could vectorize this with numpy.dot for batch
operations
                similarity = 1 - cosine(source_doc.vector, target_doc.vector)
                return max(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Causal
Extractor._calculate_semantic_distance").get("auto_param_L1300_27", 0.0), min(get_paramete
r_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._calculate_semantic_distance"
).get("auto_param_L1300_36", 1.0), similarity))

            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtracto
r._calculate_semantic_distance").get("auto_param_L1302_19", 0.5)
        except Exception:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtracto
r._calculate_semantic_distance").get("auto_param_L1304_19", 0.5)

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

        # Define transition probabilities based on logical flow
        # programa → producto → resultado → impacto
        transition_priors = {
            ('programa', 'producto'): get_parameter_loader().get("saaaaaa.analysis.derek_b
each.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1315_38", 0.85),
            ('producto', 'resultado'): get_parameter_loader().get("saaaaaa.analysis.derek_
beach.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1316_39", 0.80),
            ('resultado', 'impacto'): get_parameter_loader().get("saaaaaa.analysis.derek_b
each.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1317_38", 0.75),
            ('programa', 'resultado'): get_parameter_loader().get("saaaaaa.analysis.derek_
beach.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1318_39", 0.60),
            ('producto', 'impacto'): get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1319_37", 0.50),
            ('programa', 'impacto'): get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.CausalExtractor._calculate_type_transition_prior").get("auto_param_L1320_37", 0.30),
        }

        # Reverse transitions are less likely
        reverse_key = (target_type, source_type)
        if reverse_key in transition_priors:
            return transition_priors[reverse_key] * get_parameter_loader().get("saaaaaa.an
alysis.derek_beach.CausalExtractor._calculate_type_transition_prior").get("auto_param_L132
6_52", 0.3)

        return transition_priors.get((source_type, target_type), get_parameter_loader().ge
t("saaaaaa.analysis.derek_beach.CausalExtractor._calculate_type_transition_prior").get("au
to_param_L1328_65", 0.40))

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

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

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

        # Define causal hierarchy levels (following TeoriaCambio axioms)
```

```
        # Lower levels cannot causally influence higher levels
        hierarchy_levels = {
            'programa': 1,
            'producto': 2,
            'resultado': 3,
            'impacto': 4
        }

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

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

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

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

        return None

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

        Harmonic Front 3 - Enhancement 4: Language Specificity Assessment
        Enhanced to check policy-specific vocabulary (patrones_verificacion) for current
        Policy Area (P1–P10), not just generic causal keywords.

        For D6-Q5 (Contextual/Differential Focus): rewards use of specialized terminology
        that anchors intervention in social/cultural context (e.g., "catastro
multipropósito",
        "reparación integral", "mujeres rurales", "guardia indígena").
        """
        # Strong causal indicators
        strong_indicators = ['causa', 'produce', 'genera', 'resulta en', 'conduce a']
        # Moderate indicators
        moderate_indicators = ['permite', 'contribuye', 'facilita', 'mediante', 'a través
de']
        # Weak indicators
        weak_indicators = ['con el fin de', 'para', 'porque']

        keyword_lower = keyword.lower()

        # Base score from causal indicators
        base_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtrac
tor._calculate_language_specificity").get("base_score", 0.6) # Refactored
        if any(ind in keyword_lower for ind in strong_indicators):
            base_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalEx
tractor._calculate_language_specificity").get("base_score", 0.9) # Refactored
        elif any(ind in keyword_lower for ind in moderate_indicators):
            base_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalEx
tractor._calculate_language_specificity").get("base_score", 0.7) # Refactored
        elif any(ind in keyword_lower for ind in weak_indicators):
            base_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalEx
tractor._calculate_language_specificity").get("base_score", 0.5) # Refactored

        # HARMONIC FRONT 3 - Enhancement 4: Policy-specific vocabulary boost
        # Check for specialized terminology per policy area
```

```
        policy_area_vocabulary = {
            'P1': [  # Ordenamiento Territorial
                'catastro multipropósito', 'pot', 'pbot', 'eot', 'uaf', 'suelo de
protección',
                'zonificación', 'uso del suelo', 'densificación', 'expansión urbana'
            ],
            'P2': [  # Víctimas y Paz
                'reparación integral', 'restitución de tierras', 'víctimas del conflicto',
                'desplazamiento forzado', 'despojo', 'acción integral', 'enfoque
diferencial étnico',
                'construcción de paz', 'reconciliación', 'memoria histórica'
            ],
            'P3': [  # Desarrollo Rural
                'mujeres rurales', 'extensión agropecuaria', 'asistencia técnica rural',
                'adecuación de tierras', 'comercialización campesina', 'economía
campesina',
                'soberanía alimentaria', 'fondo de tierras'
            ],
            'P4': [  # Grupos Étnicos
                'guardia indígena', 'guardia cimarrona', 'territorios colectivos',
                'autoridades ancestrales', 'consulta previa', 'consentimiento libre',
                'medicina tradicional', 'sistema de salud propio indígena', 'jurisdicción
especial indígena'
            ],
            'P5': [  # Infraestructura y Conectividad
                'terciarias', 'vías terciarias', 'transporte intermodal', 'último
kilómetro',
                'conectividad digital', 'internet rural', 'electrificación rural'
            ],
            'P6': [  # Salud Rural
                'red hospitalaria', 'atención primaria', 'promotores de salud',
                'prevención de enfermedades tropicales', 'saneamiento básico', 'agua
segura'
            ],
            'P7': [  # Educación Rural
                'escuela nueva', 'modelos flexibles', 'post-primaria rural',
                'educación propia', 'alfabetización', 'deserción escolar rural'
            ],
            'P8': [  # Vivienda y Habitabilidad
                'mejoramiento de vivienda rural', 'materiales locales', 'construcción
sostenible',
                'vivienda de interés social rural', 'titulación predial'
            ],
            'P9': [  # Medio Ambiente
                'páramos', 'humedales', 'áreas protegidas', 'corredores biológicos',
                'servicios ecosistémicos', 'pago por servicios ambientales', 'restauración
 ecológica'
            ],
            'P10': [  # Reactivación Económica
                'encadenamientos productivos', 'economía solidaria', 'cooperativas',
                'microcrédito', 'emprendimiento asociativo', 'fondo rotatorio'
            ]
        }

        # General contextual/differential focus vocabulary (D6-Q5)
        contextual_vocabulary = [
            'enfoque diferencial', 'enfoque de género', 'enfoque étnico',
            'acción sin daño', 'pertinencia cultural', 'contexto territorial',
            'restricciones territoriales', 'barreras culturales', 'inequidad',
            'discriminación', 'exclusión', 'vulnerabilidad', 'marginalidad',
            'ruralidad dispersa', 'aislamiento geográfico', 'baja densidad poblacional',
            'población dispersa', 'difícil acceso'
        ]

        # Check for policy-specific vocabulary boost
        specificity_boost = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Causa
lExtractor._calculate_language_specificity").get("specificity_boost", 0.0) # Refactored
        text_to_check = (keyword_lower + ' ' + (context or '')).lower()
```

```
        if policy_area and policy_area in policy_area_vocabulary:
            for term in policy_area_vocabulary[policy_area]:
                if term.lower() in text_to_check:
                    specificity_boost = max(specificity_boost, get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.CausalExtractor._calculate_language_specificity").get("auto_
param_L1467_63", 0.15))
                    self.logger.debug(f"Policy-specific term detected: '{term}' for
{policy_area}")
                    break

        # Check for general contextual vocabulary (D6-Q5)
        for term in contextual_vocabulary:
            if term.lower() in text_to_check:
                specificity_boost = max(specificity_boost, get_parameter_loader().get("saa
aaaa.analysis.derek_beach.CausalExtractor._calculate_language_specificity").get("auto_para
m_L1474_59", 0.10))
                self.logger.debug(f"Contextual term detected: '{term}'")
                break

        final_score = min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalE
xtractor._calculate_language_specificity").get("auto_param_L1478_26", 1.0), base_score +
specificity_boost)

        return final_score

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

        if not source_node or not target_node:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtracto
r._assess_temporal_coherence").get("auto_param_L1489_19", 0.5)

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

            # Define logical verb sequences
            verb_sequences = {
                'diagnosticar': 1, 'planificar': 2, 'ejecutar': 3, 'evaluar': 4,
                'diseñar': 2, 'implementar': 3, 'monitorear': 4
            }

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

            if source_seq < target_seq:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_temporal_coherence").get("auto_param_L1506_23", 0.85)
            elif source_seq == target_seq:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_temporal_coherence").get("auto_param_L1508_23", 0.60)
            else:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_temporal_coherence").get("auto_param_L1510_23", 0.30)

        return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._a
ssess_temporal_coherence").get("auto_param_L1512_15", 0.50)

    @calibrated_method("saaaaaa.analysis.derek_beach.CausalExtractor._assess_financial_con
sistency")
    def _assess_financial_consistency(self, source: str, target: str) -> float:
        """Assess financial alignment between connected nodes"""
```

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

        if not source_node or not target_node:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtracto
r._assess_financial_consistency").get("auto_param_L1521_19", 0.5)

        source_budget = source_node.financial_allocation
        target_budget = target_node.financial_allocation

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

            if get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._a
ssess_financial_consistency").get("auto_param_L1530_15", 0.1) <= ratio <= get_parameter_lo
ader().get("saaaaaa.analysis.derek_beach.CausalExtractor._assess_financial_consistency").g
et("auto_param_L1530_31", 1.0):
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_financial_consistency").get("auto_param_L1531_23", 0.85)
            elif ratio > get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalEx
tractor._assess_financial_consistency").get("auto_param_L1532_25", 1.0) and ratio <= 1.5:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_financial_consistency").get("auto_param_L1533_23", 0.60)
            else:
                return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtr
actor._assess_financial_consistency").get("auto_param_L1535_23", 0.30)

        return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._a
ssess_financial_consistency").get("auto_param_L1537_15", 0.50)

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

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

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

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

        return get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._c
alculate_textual_proximity").get("auto_param_L1560_15", 0.5)

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

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

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

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

        return prior_mean, adjusted_alpha, adjusted_beta

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

        Enhanced with:
        - Nonlinear transformation rewarding triangulation
        - Evidence diversity verification across analytical domains
        """
        # Weight different evidence types
        weights = {
            'semantic_distance': get_parameter_loader().get("saaaaaa.analysis.derek_beach.
CausalExtractor._calculate_composite_likelihood").get("auto_param_L1591_33", 0.25),
            'type_transition_prior': get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1592_37", 0.20),
            'language_specificity': get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1593_36", 0.20),
            'temporal_coherence': get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1594_34", 0.15),
            'financial_consistency': get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1595_37", 0.10),
            'textual_proximity': get_parameter_loader().get("saaaaaa.analysis.derek_beach.
CausalExtractor._calculate_composite_likelihood").get("auto_param_L1596_33", 0.10)
        }

        # Basic weighted average
        likelihood = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtrac
tor._calculate_composite_likelihood").get("likelihood", 0.0) # Refactored
        evidence_count = 0
        domain_diversity = set()

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

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

        # Triangulation bonus: Exponentially reward multiple independent observations
        # D6-Q4/Q5 (Adaptiveness/Context) - evidence across different analytical domains
        diversity_count = len(domain_diversity)
        if diversity_count >= 3:
            # Strong triangulation across semantic, temporal, and financial domains
            triangulation_bonus = get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1624_34", 1.0) + get_p
arameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._calculate_composite_l
ikelihood").get("auto_param_L1624_40", 0.15) * np.exp(diversity_count - 2)
        elif diversity_count == 2:
            # Moderate triangulation
            triangulation_bonus = get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1627_34", 1.0)5
```

```
        else:
            # Weak or no triangulation
            triangulation_bonus = get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._calculate_composite_likelihood").get("triangulation_bonus", 1.0) #
Refactored

        # Apply nonlinear transformation
        enhanced_likelihood = min(get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1633_34", 1.0),
likelihood * triangulation_bonus)

        # Penalty for insufficient evidence diversity
        if evidence_count < 3:
            enhanced_likelihood *= get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.CausalExtractor._calculate_composite_likelihood").get("auto_param_L1637_35", 0.85)

        return enhanced_likelihood


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

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

        # Connect productos to programas
        for prod in nodes_by_type.get('producto', []):
            for prog in nodes_by_type.get('programa', []):
                if not self.graph.has_edge(prog, prod):
                    self.graph.add_edge(prog, prod, logic='inferido', strength=get_paramet
er_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._build_type_hierarchy").get(
"auto_param_L1654_79", 0.5))

        # Connect resultados to productos
        for res in nodes_by_type.get('resultado', []):
            for prod in nodes_by_type.get('producto', []):
                if not self.graph.has_edge(prod, res):
                    self.graph.add_edge(prod, res, logic='inferido', strength=get_paramete
r_loader().get("saaaaaa.analysis.derek_beach.CausalExtractor._build_type_hierarchy").get("
auto_param_L1660_78", 0.5))


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

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

        Returns:
            Confidence score between 0 and 1
        """
        confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtrac
tor._calculate_confidence").get("confidence", 0.5) # Refactored

        # Increase confidence if node has quantitative targets
        if node.target and node.baseline:
            try:
                float(str(node.target).replace(',', '').replace('%', ''))
                confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Cau
salExtractor._calculate_confidence").get("auto_param_L1680_30", 0.2)
            except (ValueError, TypeError):
                pass
```

```
        # Increase confidence if text has causal indicators
        if link_text:
            causal_words = ['porque', 'debido', 'mediante', 'a través', 'permite',
'genera', 'produce']
            if any(word in link_text.lower() for word in causal_words):
                confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Cau
salExtractor._calculate_confidence").get("auto_param_L1688_30", 0.15)

        # Increase confidence based on rigor status
        if hasattr(node, 'rigor_status'):
            if node.rigor_status == 'fuerte':
                confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Cau
salExtractor._calculate_confidence").get("auto_param_L1693_30", 0.15)
            elif node.rigor_status == 'débil':
                confidence -= get_parameter_loader().get("saaaaaa.analysis.derek_beach.Cau
salExtractor._calculate_confidence").get("auto_param_L1695_30", 0.1)

        return min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.CausalExtracto
r._calculate_confidence").get("auto_param_L1697_19", 1.0), max(get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.CausalExtractor._calculate_confidence").get("auto_param_L169
7_28", 0.0), confidence))

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

        Args:
            text: Goal text to classify

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

        # Keywords for each type
        if any(word in text_lower for word in ['programa', 'línea estratégica',
'componente', 'eje']):
            return 'programa'
        elif any(word in text_lower for word in ['producto', 'servicio', 'bien',
'actividad']):
            return 'producto'
        elif any(word in text_lower for word in ['resultado', 'efecto', 'cambio',
'mejora']):
            return 'resultado'
        elif any(word in text_lower for word in ['impacto', 'transformación',
'desarrollo', 'bienestar']):
            return 'impacto'
        # Default classification based on position and complexity
        elif len(text) < 100:
            return 'producto'
        else:
            return 'resultado'

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

        Args:
            text: Text to extract justifications from

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

```
        # Patterns that indicate causal justifications
        patterns = [
            r'porque\s+([^.]+)',
            r'debido\s+a\s+([^.]+)',
            r'mediante\s+([^.]+)',
            r'a\s+través\s+de\s+([^.]+)',
            r'se\s+logra\s+mediante\s+([^.]+)',
            r'permite\s+([^.]+)',
            r'genera\s+([^.]+)',
        ]

        for pattern in patterns:
            matches = re.finditer(pattern, text, re.IGNORECASE)
            for match in matches:
                justification_text = match.group(1).strip()
                justifications.append({
                    'text': justification_text,
                    'confidence': get_parameter_loader().get("saaaaaa.analysis.derek_beach
.CausalExtractor._extract_causal_justifications").get("auto_param_L1757_34", 0.7),
                    'type': 'causal_explanation'
                })

        return justifications

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

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

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

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

        if not main_verb:
            return None

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

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

        if entity and main_verb:
            return EntityActivity(
                entity=entity,
                activity=main_verb.text,
```

```
                verb_lemma=main_verb.lemma_,
                confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Mech
anismPartExtractor.extract_entity_activity").get("confidence", 0.85) # Refactored
            )

        return None

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

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

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

        Returns:
            Confidence score between 0 and 1
        """
        confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.MechanismPar
tExtractor._calculate_ea_confidence").get("confidence", 0.5) # Refactored

        # Higher confidence if entity is in known aliases
        if entity.upper() in self.entity_aliases:
            confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Mechani
smPartExtractor._calculate_ea_confidence").get("auto_param_L1834_26", 0.2)

        # Higher confidence if activity is a strong verb
        strong_verbs = ['ejecutar', 'implementar', 'desarrollar', 'gestionar',
'coordinar']
        if any(verb in activity.lower() for verb in strong_verbs):
            confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Mechani
smPartExtractor._calculate_ea_confidence").get("auto_param_L1839_26", 0.15)

        # Higher confidence if there's clear grammatical connection in context
        if entity in context and activity in context:
            confidence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Mechani
smPartExtractor._calculate_ea_confidence").get("auto_param_L1843_26", 0.15)

        return min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.MechanismPartE
xtractor._calculate_ea_confidence").get("auto_param_L1845_19", 1.0), confidence)

    @calibrated_method("saaaaaa.analysis.derek_beach.MechanismPartExtractor._find_action_v
erb")
    def _find_action_verb(self, text: str) -> str | None:
        """
        Find the main action verb in text.

        Args:
            text: Text to analyze

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

        # Find main verb
        for token in doc:
```

```
            if token.pos_ == 'VERB' and token.dep_ in ['ROOT', 'ccomp', 'xcomp']:
                return token.text

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

        return None

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

        Args:
            text: Text to analyze

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

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

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

        return None

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

        Args:
            entity: Entity text
            activity: Activity text

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

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

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

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

    def __init__(self, config: ConfigLoader) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.config = config
        self.financial_data: dict[str, dict[str, float]] = {}
        self.unit_costs: dict[str, float] = {}
```

```
        self.successful_parses = 0
        self.failed_parses = 0
        self.d3_q3_analysis: dict[str, Any] = {}  # Harmonic Front 3 - D3-Q3 metrics

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

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

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

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

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

        amount_col = None
        program_col = None

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

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

```
            if program_pattern.search(val_str) and not program_col:
                program_col = i
                table.columns = table.iloc[0]
                table = table[1:]

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

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

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

                        # Update node
                        nodes[matched_node].financial_allocation = amount

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

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

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

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


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

        Enhanced for D1-Q3 / D3-Q3 Financial Traceability:
        - Implements confidence penalty if fuzzy match ratio < 100
        - Reduces node.financial_allocation confidence by 15% for imperfect matches
```

```
        - Tracks match quality for overall financial traceability scoring
        """
        if program_id in nodes:
            # Perfect match - no penalty
            return program_id

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

        if best_match:
            matched_node_id = best_match[0]
            match_ratio = best_match[1]

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

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

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

                # Store match confidence for D1-Q3 / D3-Q3 scoring
                if not hasattr(node, 'financial_match_confidence'):
                    node.financial_match_confidence = match_ratio / 10get_parameter_loader
().get("saaaaaa.analysis.derek_beach.FinancialAuditor._match_program_to_node").get("auto_p
aram_L2098_70", 0.0)
                else:
                    # Average if multiple matches
                    node.financial_match_confidence = (node.financial_match_confidence +
match_ratio / 10get_parameter_loader().get("saaaaaa.analysis.derek_beach.FinancialAuditor.
_match_program_to_node").get("auto_param_L2101_105", 0.0)) / 2

            return matched_node_id

        return None

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

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

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

```
        d3_q3_scores = {}

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

            # Check if node has financial allocation
            has_budget = node.financial_allocation is not None and
node.financial_allocation > 0

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

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

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

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

            if has_budget and has_mechanism:
                necessity_score += get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2147_35", 0.40)
 # Budget + mechanism present

            if has_budget and has_dependencies:
                necessity_score += get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2150_35", 0.30)
 # Budget supports downstream goals

            if is_specific_allocation:
                necessity_score += get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2153_35", 0.30)
 # Specific allocation (not generic)

            # D3-Q3 quality criteria
            d3_q3_quality = 'insuficiente'
            if necessity_score >= get_parameter_loader().get("saaaaaa.analysis.derek_beach
.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2157_34", 0.85):
                d3_q3_quality = 'excelente'
            elif necessity_score >= get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2159_36",
0.70):
                d3_q3_quality = 'bueno'
            elif necessity_score >= get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2161_36",
0.50):
                d3_q3_quality = 'aceptable'

            d3_q3_scores[node_id] = {
                'necessity_score': necessity_score,
                'd3_q3_quality': d3_q3_quality,
                'has_budget': has_budget,
                'has_mechanism': has_mechanism,
                'has_dependencies': has_dependencies,
                'is_specific_allocation': is_specific_allocation,
```

```
            'counterfactual_sufficient': necessity_score < get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check").get(
"auto_param_L2171_63", 0.50),  # Would still execute without budget
            'budget_necessary': necessity_score >= get_parameter_loader().get("saaaaaa
.analysis.derek_beach.FinancialAuditor._perform_counterfactual_budget_check").get("auto_pa
ram_L2172_55", 0.70)  # Budget is necessary
        }

        # Store in node for later retrieval
        node.audit_flags = node.audit_flags or []
        if necessity_score < get_parameter_loader().get("saaaaaa.analysis.derek_beach.
FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2177_33", 0.50):
            node.audit_flags.append('budget_not_necessary')
            self.logger.warning(
                f"D3-Q3: {node_id} may execute without allocated budget
(score={necessity_score:.2f})")
        elif necessity_score >= get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.FinancialAuditor._perform_counterfactual_budget_check").get("auto_param_L2181_36",
0.85):
            node.audit_flags.append('budget_well_traced')
            self.logger.info(f"D3-Q3: {node_id} has well-traced, necessary budget
(score={necessity_score:.2f})")

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

        1)
    }

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


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

    Args:
        allocation: Financial allocation amount
        target: Target value

    Returns:
        Sufficiency ratio (get_parameter_loader().get("saaaaaa.analysis.derek_beach.Fi
nancialAuditor._calculate_sufficiency").get("auto_param_L2208_31", 1.0) = exactly
sufficient, >get_parameter_loader().get("saaaaaa.analysis.derek_beach.FinancialAuditor._ca
lculate_sufficiency").get("auto_param_L2208_58", 1.0) = oversufficient)
    """
    if not target or target == 0:
        return get_parameter_loader().get("saaaaaa.analysis.derek_beach.FinancialAudit
or._calculate_sufficiency").get("auto_param_L2211_19", 0.0)

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

    # Compare with historical/expected unit costs if available
    # For now, return simple ratio
    return allocation / target if target > 0 else get_parameter_loader().get("saaaaaa.
analysis.derek_beach.FinancialAuditor._calculate_sufficiency").get("auto_param_L2218_54",
0.0)
```

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

        Args:
            nodes: Dictionary of nodes

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

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

            # Check for insufficient allocation
            if node.financial_allocation and node.target:
                try:
                    target_val = float(str(node.target).replace(',', '').replace('%', ''))
                    if target_val > 0:
                        sufficiency =
self._calculate_sufficiency(node.financial_allocation, target_val)
                        if sufficiency < get_parameter_loader().get("saaaaaa.analysis.dere
k_beach.FinancialAuditor._detect_allocation_gaps").get("auto_param_L2249_41", 0.5):
                            gaps.append({
                                'node_id': node_id,
                                'type': 'insufficient_allocation',
                                'severity': 'medium',
                                'message': f"Low sufficiency ratio {sufficiency:.2f} for
{node_id}",
                                'sufficiency': sufficiency
                            })
                except (ValueError, TypeError):
                    pass

        return gaps


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

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

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

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

        best_match = None
```

```
        best_score = 0

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

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

            if score > best_score and score > get_parameter_loader().get("saaaaaa.analysis
.derek_beach.FinancialAuditor._match_goal_to_budget").get("auto_param_L2291_46", 0.3):  #
Minimum threshold
                best_score = score
                best_match = entry

        return best_match

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

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

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

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

        producto_nodes_count = 0
        producto_nodes_passed = 0

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

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

            # Extract quantitative claims from node text if detector available
            quantitative_claims = []
            if has_detector:
                try:
                    # Create temporary detector instance
```

```
                detector = PolicyContradictionDetectorV2(device='cpu')
                quantitative_claims =
detector._extract_structured_quantitative_claims(node.text)
            except Exception as e:
                self.logger.debug(f"Could not extract quantitative claims: {e}")

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

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

        # D3-Q1 Ficha Técnica compliance check for producto nodes
        if node.type == 'producto':
            # Check if has all minimum DNP INDICATOR_STRUCTURE elements
            has_complete_ficha = (
                    baseline_valid and
                    target_valid and
                    'sin_linea_base' not in node.audit_flags and
                    'sin_meta' not in node.audit_flags
            )

            if has_complete_ficha and quantitative_claims:
                # Node passes D3-Q1 compliance
                producto_nodes_passed += 1
                result['recommendations'].append(f"D3-Q1 Ficha Técnica completa para
{node_id}")
            elif has_complete_ficha:
                # Has baseline/target but no quantitative claims verification
                producto_nodes_passed += get_parameter_loader().get("saaaaaa.analysis.
derek_beach.OperationalizationAuditor.audit_evidence_traceability").get("auto_param_L2401_
45", 0.5)  # Partial credit
```

```
                result['warnings'].append(f"D3-Q1 parcial: Ficha básica sin
verificación cuantitativa en {node_id}")

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

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

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

            self.audit_results[node_id] = result

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

            if d3_q1_compliance_pct >= 80:
                self.logger.info("D3-Q1 Score: EXCELENTE (≥80% productos con Ficha Técnica
 completa)")
            elif d3_q1_compliance_pct >= 60:
                self.logger.info("D3-Q1 Score: BUENO (60-80% compliance)")
            else:
                self.logger.warning("D3-Q1 Score: INSUFICIENTE (<60% compliance)")

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

        return self.audit_results

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

        # Group nodes by program
        programs: dict[str, list[str]] = defaultdict(list)
        for node_id in graph.nodes():
            node_data = graph.nodes[node_id]
            if node_data.get('type') == 'programa':
                for successor in graph.successors(node_id):
                    if graph.nodes[successor].get('type') == 'producto':
                        programs[node_id].append(successor)

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

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

```
                sequence_num = self.verb_sequences.get(verb, 999)
                activities.append((goal_id, verb, sequence_num))

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

    self.sequence_warnings = warnings
    return warnings

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

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

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

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

    # Audit results by layers
    layer1_results = self._audit_direct_evidence(nodes, scm_dag, historical_data)
    layer2_results = self._audit_causal_implications(nodes, graph, layer1_results)
    layer3_results = self._audit_systemic_risk(nodes, graph, layer1_results,
layer2_results, pdet_alignment)

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

    audit_report = {
        'direct_evidence': layer1_results,
        'causal_implications': layer2_results,
        'systemic_risk': layer3_results,
        'recommendations': recommendations,
        'summary': {
            'total_nodes': len(nodes),
            'critical_omissions': sum(1 for r in layer1_results.values()
                                      if r.get('omission_severity') == 'critical'),
            'expected_success_probability': layer3_results.get('success_probability',
get_parameter_loader().get("saaaaaa.analysis.derek_beach.OperationalizationAuditor.bayesia
n_counterfactual_audit").get("auto_param_L2518_90", 0.0)),
            'risk_score': layer3_results.get('risk_score', get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.OperationalizationAuditor.bayesian_counterfactual_audit").ge
t("auto_param_L2519_63", 0.0))
        }
    }
```

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

        return audit_report

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

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

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

        return dag

    @calibrated_method("saaaaaa.analysis.derek_beach.OperationalizationAuditor._get_defaul
t_historical_priors")
    def _get_default_historical_priors(self) -> dict[str, Any]:
        """Get default historical priors if no data is available"""
        return {
            'entity_presence_success_rate': get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_param_L255
6_44", 0.94),
            'baseline_presence_success_rate': get_parameter_loader().get("saaaaaa.analysis
.derek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_param_L2
557_46", 0.89),
            'target_presence_success_rate': get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_param_L255
8_44", 0.92),
            'budget_presence_success_rate': get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_param_L255
9_44", 0.78),
            'mechanism_presence_success_rate': get_parameter_loader().get("saaaaaa.analysi
s.derek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_param_L
2560_47", 0.65),
            'complete_documentation_success_rate': get_parameter_loader().get("saaaaaa.ana
lysis.derek_beach.OperationalizationAuditor._get_default_historical_priors").get("auto_par
am_L2561_51", 0.82),
            'node_type_success_rates': {
                'producto': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Opera
tionalizationAuditor._get_default_historical_priors").get("auto_param_L2563_28", 0.85),
                'resultado': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Oper
ationalizationAuditor._get_default_historical_priors").get("auto_param_L2564_29", 0.72),
                'impacto': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Operat
ionalizationAuditor._get_default_historical_priors").get("auto_param_L2565_27", 0.58)
            }
        }

    @calibrated_method("saaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_dire
ct_evidence")
    def _audit_direct_evidence(self, nodes: dict[str, MetaNode],
```

```
                        scm_dag: nx.DiGraph,
                        historical_data: dict[str, Any]) -> dict[str, dict[str,
Any]]:
        """Layer 1: Audit direct evidence of required components

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

        # Load highly specific priors for rare evidence types
        # D2-Q4: Risk matrices are rare in poor PDMs (high probative value as Smoking Gun)
        rare_evidence_priors = {
            'risk_matrix': {
                'prior_alpha': 1.5,  # Low alpha = rare occurrence
                'prior_beta': 12.0,  # High beta = high failure rate when absent
                'keywords': ['matriz de riesgo', 'análisis de riesgo', 'gestión de
riesgo', 'riesgos identificados']
            },
            'unwanted_effects': {
                'prior_alpha': 1.8,  # D5-Q5: Effects analysis is also rare
                'prior_beta': 1get_parameter_loader().get("saaaaaa.analysis.derek_beach.Op
erationalizationAuditor._audit_direct_evidence").get("auto_param_L2590_31", 0.5),
                'keywords': ['efectos no deseados', 'efectos adversos', 'impactos
negativos',
                             'consecuencias no previstas']
            },
            'theory_of_change': {
                'prior_alpha': 1.2,
                'prior_beta': 15.0,
                'keywords': ['teoría de cambio', 'teoría del cambio', 'cadena causal',
'modelo lógico']
            }
        }

        for node_id, node in nodes.items():
            omissions = []
            omission_probs = {}
            rare_evidence_found = {}

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

            # Check baseline
            if not node.baseline or str(node.baseline).upper() in ['ND', 'POR DEFINIR',
'N/A', 'NONE']:
                p_failure_given_omission = get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2621_43",
1.0) - historical_data.get('baseline_presence_success_rate', get_parameter_loader().get("s
aaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_p
aram_L2621_103", 0.89))
                omissions.append('baseline')
                omission_probs['baseline'] = p_failure_given_omission

            # Check target
            if not node.target or str(node.target).upper() in ['ND', 'POR DEFINIR', 'N/A',
```

```
 'NONE']:
                p_failure_given_omission = get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2627_43",
1.0) - historical_data.get('target_presence_success_rate', get_parameter_loader().get("saa
aaaa.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_par
am_L2627_101", 0.92))
                omissions.append('target')
                omission_probs['target'] = p_failure_given_omission

            # Check entity
            if not node.responsible_entity:
                p_failure_given_omission = get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2633_43",
1.0) - historical_data.get('entity_presence_success_rate', get_parameter_loader().get("saa
aaaa.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_par
am_L2633_101", 0.94))
                omissions.append('entity')
                omission_probs['entity'] = p_failure_given_omission

            # Check budget
            if not node.financial_allocation:
                p_failure_given_omission = get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2639_43",
1.0) - historical_data.get('budget_presence_success_rate', get_parameter_loader().get("saa
aaaa.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_par
am_L2639_101", 0.78))
                omissions.append('budget')
                omission_probs['budget'] = p_failure_given_omission

            # Check mechanism
            if not node.entity_activity:
                p_failure_given_omission = get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2645_43",
1.0) - historical_data.get('mechanism_presence_success_rate', get_parameter_loader().get("
saaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_
param_L2645_104", 0.65))
                omissions.append('mechanism')
                omission_probs['mechanism'] = p_failure_given_omission

            # Determine severity
            severity = 'none'
            if omission_probs:
                max_failure_prob = max(omission_probs.values())
                if max_failure_prob > get_parameter_loader().get("saaaaaa.analysis.derek_b
each.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2653_38", 0.15):
                    severity = 'critical'
                elif max_failure_prob > get_parameter_loader().get("saaaaaa.analysis.derek
_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2655_40",
0.10):
                    severity = 'high'
                elif max_failure_prob > get_parameter_loader().get("saaaaaa.analysis.derek
_beach.OperationalizationAuditor._audit_direct_evidence").get("auto_param_L2657_40",
0.05):
                    severity = 'medium'
                else:
                    severity = 'low'

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

        return results

    @calibrated_method("saaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_caus
```

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

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

            # If baseline is missing
            if 'baseline' in node_omissions:
                # P(target_miscalibrated | missing_baseline)
                causal_effects['target_miscalibration'] = {
                    'probability': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.OperationalizationAuditor._audit_causal_implications").get("auto_param_L2687_35", 0.73),
                    'description': 'Sin línea base, la meta probablemente está mal
calibrada'
                }

            # If entity and high budget are missing
            if 'entity' in node_omissions and node.financial_allocation and
node.financial_allocation > 1000000:
                causal_effects['implementation_failure'] = {
                    'probability': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.OperationalizationAuditor._audit_causal_implications").get("auto_param_L2694_35", 0.89),
                    'description': 'Alto presupuesto sin entidad responsable indica alto
riesgo de falla'
                }
            elif 'entity' in node_omissions:
                causal_effects['implementation_failure'] = {
                    'probability': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.OperationalizationAuditor._audit_causal_implications").get("auto_param_L2699_35", 0.65),
                    'description': 'Sin entidad responsable, la implementación es
incierta'
                }

            # If mechanism is missing
            if 'mechanism' in node_omissions:
                causal_effects['unclear_pathway'] = {
                    'probability': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.OperationalizationAuditor._audit_causal_implications").get("auto_param_L2706_35", 0.70),
                    'description': 'Sin mecanismo definido, la vía causal es opaca'
                }

            # Check downstream effects
            successors = list(graph.successors(node_id)) if graph.has_node(node_id) else
[]
            if node_omissions and successors:
                causal_effects['cascade_risk'] = {
                    'probability': min(get_parameter_loader().get("saaaaaa.analysis.derek_
beach.OperationalizationAuditor._audit_causal_implications").get("auto_param_L2714_39",
0.95), get_parameter_loader().get("saaaaaa.analysis.derek_beach.OperationalizationAuditor.
_audit_causal_implications").get("auto_param_L2714_45", 0.4) + get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_causal_implications").get("
auto_param_L2714_51", 0.1) * len(node_omissions)),
                    'affected_nodes': successors,
                    'description': f'Omisiones pueden afectar {len(successors)} nodos
dependientes'
                }

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

```
        return implications

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

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

        For D5-Q4 (Riesgos Sistémicos) and D4-Q5 (Alineación):
        - If pdet_alignment ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ope
rationalizationAuditor._audit_systemic_risk").get("auto_param_L2740_30", 0.60), applies
1.2× multiplier to risk_score
        - Excelente on D5-Q4 requires risk_score < get_parameter_loader().get("saaaaaa.ana
lysis.derek_beach.OperationalizationAuditor._audit_systemic_risk").get("auto_param_L2741_5
1", 0.10)

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

        # Identify critical nodes (high centrality)
        if graph.number_of_nodes() > 0:
            try:
                centrality = nx.betweenness_centrality(graph)
            except (nx.NetworkXError, ZeroDivisionError, Exception) as e:
                logging.warning(f"Failed to calculate betweenness centrality: {e}. Using
default values.")
                centrality = dict.fromkeys(graph.nodes(), get_parameter_loader().get("saaa
aaa.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk").get("auto_param_
L2752_58", 0.5))
        else:
            centrality = {}

        # Calculate P(cascade_failure | omission_set)
        critical_omissions = []
        for node_id, evidence in direct_evidence.items():
            if evidence['omission_severity'] in ['critical', 'high']:
                node_centrality = centrality.get(node_id, get_parameter_loader().get("saaa
aaa.analysis.derek_beach.OperationalizationAuditor._audit_systemic_risk").get("auto_param_
L2760_58", 0.5))
                critical_omissions.append({
                    'node_id': node_id,
                    'severity': evidence['omission_severity'],
                    'centrality': node_centrality,
                    'omissions': evidence['omissions']
                })

        # Calculate systemic risk
        if critical_omissions:
            # Weighted by centrality
            risk_score = sum(
                (get_parameter_loader().get("saaaaaa.analysis.derek_beach.Operationalizati
onAuditor._audit_systemic_risk").get("auto_param_L2772_17", 1.0) if om['severity'] ==
'critical' else get_parameter_loader().get("saaaaaa.analysis.derek_beach.Operationalizatio
nAuditor._audit_systemic_risk").get("auto_param_L2772_58", 0.7)) * (om['centrality'] + get
_parameter_loader().get("saaaaaa.analysis.derek_beach.OperationalizationAuditor._audit_sys
temic_risk").get("auto_param_L2772_85", 0.1))
                for om in critical_omissions
            ) / len(nodes)
        else:
```

```
            risk_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Operatio
nalizationAuditor._audit_systemic_risk").get("risk_score", 0.0) # Refactored

        # AUDIT POINT 2.3: Policy Alignment Dual Constraint
        # If pdet_alignment ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ope
rationalizationAuditor._audit_systemic_risk").get("auto_param_L2779_30", 0.60), apply 1.2×
 multiplier to risk_score
        # This enforces integration between D4-Q5 (Alineación) and D5-Q4 (Riesgos
Sistémicos)
        alignment_penalty_applied = False
        alignment_threshold = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ope
rationalizationAuditor._audit_systemic_risk").get("alignment_threshold", 0.6) # Refactored
        alignment_multiplier = 1.2

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

        # Calculate P(success | current_state)
        total_omissions = sum(len(e['omissions']) for e in direct_evidence.values())
        total_possible = len(nodes) * 5  # 5 key attributes per node
        completeness = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Operationa
lizationAuditor._audit_systemic_risk").get("auto_param_L2798_23", 1.0) - (total_omissions
/ max(total_possible, 1))

        # Success probability (simplified Bayesian)
        base_success_rate = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Opera
tionalizationAuditor._audit_systemic_risk").get("base_success_rate", 0.7) # Refactored
        success_probability = base_success_rate * completeness

        # D5-Q4 quality criteria check (AUDIT POINT 2.3)
        # Excellent requires risk_score < get_parameter_loader().get("saaaaaa.analysis.der
ek_beach.OperationalizationAuditor._audit_systemic_risk").get("auto_param_L2805_42", 0.10)
 (matching ODS benchmarks per UN 2020)
        d5_q4_quality = 'insuficiente'
        risk_threshold_excellent = get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.OperationalizationAuditor._audit_systemic_risk").get("risk_threshold_excellent", 0.1) #
Refactored
        risk_threshold_good = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ope
rationalizationAuditor._audit_systemic_risk").get("risk_threshold_good", 0.2) # Refactored
        risk_threshold_acceptable = get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.OperationalizationAuditor._audit_systemic_risk").get("risk_threshold_acceptable", 0.35)
 # Refactored

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

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

        return {
            'risk_score': min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ope
```

```
rationalizationAuditor._audit_systemic_risk").get("auto_param_L2826_30", 1.0),
risk_score),
            'success_probability': success_probability,
            'critical_omissions': critical_omissions,
            'completeness': completeness,
            'total_omissions': total_omissions,
            'pdet_alignment': pdet_alignment,
            'alignment_penalty_applied': alignment_penalty_applied,
            'alignment_threshold': alignment_threshold,
            'alignment_multiplier': alignment_multiplier,
            'alignment_causing_failure': alignment_causing_failure,
            'd5_q4_quality': d5_q4_quality,
            'd4_q5_alignment_score': pdet_alignment,
            'risk_thresholds': {
                'excellent': risk_threshold_excellent,
                'good': risk_threshold_good,
                'acceptable': risk_threshold_acceptable
            }
        }

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

        # Calculate expected value of information for each remediation
        for node_id, evidence in direct_evidence.items():
            if not evidence['omissions']:
                continue

            for omission in evidence['omissions']:
                # Estimate impact
                omission_prob = evidence['omission_probabilities'].get(omission, get_param
eter_loader().get("saaaaaa.analysis.derek_beach.OperationalizationAuditor._generate_optima
l_remediations").get("auto_param_L2860_81", 0.1))
                causal_risk = causal_implications[node_id]['total_risk']

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

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

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

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

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

        return remediations

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

    @calibrated_method("saaaaaa.analysis.derek_beach.OperationalizationAuditor._perform_co
unterfactual_budget_check")
    def _perform_counterfactual_budget_check(self, nodes: dict[str, MetaNode],
                                             graph: nx.DiGraph) -> dict[str, Any]:
        """
        Perform counterfactual budget check for operationalization audit.

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

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

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

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

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

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

        return results

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

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

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

    Legacy methods are preserved for backward compatibility.
    """

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

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

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

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

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

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

        # Typical activity sequences by mechanism type
        # These could also be externalized if needed for domain-specific customization
```

```
        self.mechanism_sequences = {
            'administrativo': ['planificar', 'coordinar', 'gestionar', 'supervisar'],
            'tecnico': ['diagnosticar', 'diseñar', 'implementar', 'evaluar'],
            'financiero': ['asignar', 'ejecutar', 'auditar', 'reportar'],
            'politico': ['concertar', 'negociar', 'aprobar', 'promulgar']
        }

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

    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianMechanismInference._log_refac
tored_components")
    def _log_refactored_components(self) -> None:
        """Log status of refactored Bayesian components (F1.2)"""
        if self.bayesian_adapter:
            status = self.bayesian_adapter.get_component_status()
            self.logger.info("  - BayesianPriorBuilder: " +
                             ("✓" if status['prior_builder_ready'] else "✗"))
            self.logger.info("  - BayesianSamplingEngine: " +
                             ("✓" if status['sampling_engine_ready'] else "✗"))
            self.logger.info("  - NecessitySufficiencyTester: " +
                             ("✓" if status['necessity_tester_ready'] else "✗"))

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

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

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

        # Track uncertainties for mean calculation
        mechanism_uncertainties = []

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

            # Track mechanism type uncertainty for quality criteria
            if 'uncertainty' in mechanism:
                mech_type_uncertainty = mechanism['uncertainty'].get('mechanism_type', get
_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference.infer_mec
hanisms").get("auto_param_L3063_87", 1.0))
                mechanism_uncertainties.append(mech_type_uncertainty)

        # Calculate mean mechanism uncertainty for Harmonic Front 4 quality criteria
        mean_mech_uncertainty = (
            np.mean(mechanism_uncertainties) if mechanism_uncertainties else get_parameter
_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference.infer_mechanisms").
get("auto_param_L3068_77", 1.0)
        )

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

        # Store for reporting
        self._mean_mechanism_uncertainty = mean_mech_uncertainty

        return self.inferred_mechanisms
```

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

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

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

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

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

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

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

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

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

        observations = {
            'entity_activity': None,
            'verbs': [],
            'entities': [],
            'budget': node.financial_allocation,
            'context_snippets': []
        }

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

        # Extract context around node mentions
```

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

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

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

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

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

        return observations

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

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

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

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

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

                    # Bayesian update
                    posterior[mech_type] *= likelihood

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

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

        return posterior

    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianMechanismInference._infer_act
ivity_sequence")
    def _infer_activity_sequence(self, observations: dict[str, Any],
                                 mechanism_type_posterior: dict[str, float]) -> dict[str,
Any]:
        """Infer activity sequence parameters"""
        # Get most likely mechanism type
```

```
        best_type = max(mechanism_type_posterior.items(), key=lambda x: x[1])[0]
        expected_sequence = self.mechanism_sequences.get(best_type, [])

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

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

            # Check if transition is observed
            if current in observed_verbs and next_verb in observed_verbs:
                transitions[(current, next_verb)] = get_parameter_loader().get("saaaaaa.an
alysis.derek_beach.BayesianMechanismInference._infer_activity_sequence").get("auto_param_L
3222_52", 0.85)
            else:
                transitions[(current, next_verb)] = get_parameter_loader().get("saaaaaa.an
alysis.derek_beach.BayesianMechanismInference._infer_activity_sequence").get("auto_param_L
3224_52", 0.40)

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

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

        # Factor 1: Entity-Activity presence
        if observations.get('entity_activity'):
            coherence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesian
MechanismInference._calculate_coherence_factor").get("auto_param_L3243_25", 0.30)
            weights.append(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._calculate_coherence_factor").get("auto_param_L3244_27", 0.30))

        # Factor 2: Budget consistency
        if observations.get('budget'):
            coherence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesian
MechanismInference._calculate_coherence_factor").get("auto_param_L3248_25", 0.20)
            weights.append(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._calculate_coherence_factor").get("auto_param_L3249_27", 0.20))

        # Factor 3: Verb sequence completeness
        seq_info = observations.get('verbs', [])
        if seq_info:
            verb_score = min(len(seq_info) / 4.0, get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.BayesianMechanismInference._calculate_coherence_factor").get("auto_param_
L3254_50", 1.0))  # Expect ~4 verbs
            coherence += verb_score * get_parameter_loader().get("saaaaaa.analysis.derek_b
each.BayesianMechanismInference._calculate_coherence_factor").get("auto_param_L3255_38",
0.25)
            weights.append(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._calculate_coherence_factor").get("auto_param_L3256_27", 0.25))

        # Factor 4: Entity presence
        if observations.get('entities'):
            coherence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesian
```

```
MechanismInference._calculate_coherence_factor").get("auto_param_L3260_25", 0.15)
            weights.append(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._calculate_coherence_factor").get("auto_param_L3261_27", 0.15))

        # Factor 5: Context richness
        snippets = observations.get('context_snippets', [])
        if snippets:
            coherence += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesian
MechanismInference._calculate_coherence_factor").get("auto_param_L3266_25", 0.10)
            weights.append(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._calculate_coherence_factor").get("auto_param_L3267_27", 0.10))

        # Normalize by actual weights used
        if weights:
            coherence = coherence / sum(weights) if sum(weights) > 0 else get_parameter_lo
ader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._calculate_coherence_f
actor").get("auto_param_L3271_74", 0.0)

        return coherence

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

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

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

        sufficiency_score = (
            (get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanis
mInference._test_sufficiency").get("auto_param_L3289_17", 0.4) if has_entity else get_para
meter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._test_sufficie
ncy").get("auto_param_L3289_40", 0.0)) +
            (get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanis
mInference._test_sufficiency").get("auto_param_L3290_17", 0.4) if has_activities else get_
parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._test_suff
iciency").get("auto_param_L3290_44", 0.0)) +
            (get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanis
mInference._test_sufficiency").get("auto_param_L3291_17", 0.2) if has_resources else get_p
arameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._test_suffi
ciency").get("auto_param_L3291_43", 0.0))
        )

        return {
            'score': sufficiency_score,
            'is_sufficient': sufficiency_score >= get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.BayesianMechanismInference._test_sufficiency").get("auto_param_L3296_50",
 0.6),
            'components': {
                'entity': has_entity,
                'activities': has_activities,
                'resources': has_resources
            }
        }

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

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

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

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

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

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

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

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

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

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

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

result = {
    'score': necessity_score,
    'is_necessary': is_necessary,
```

```
            'missing_components': missing_components,
            'alternatives_likely': not is_necessary,
            'hoop_test_passed': is_necessary
        }

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

        return result

    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianMechanismInference._generate_
necessity_remediation")
    def _generate_necessity_remediation(self, node_id: str, missing_components: list[str])
 -> str:
        """Generate remediation text for failed necessity test"""
        component_descriptions = {
            'entity': 'entidad responsable claramente identificada',
            'unique_entity': 'una única entidad responsable (múltiples entidades
detectadas)',
            'activity': 'secuencia de actividades documentada',
            'specific_activity': 'actividades específicas (no genéricas)',
            'budget': 'presupuesto asignado y cuantificado'
        }

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

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

    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianMechanismInference._quantify_
uncertainty")
    def _quantify_uncertainty(self, mechanism_type_posterior: dict[str, float],
                              sequence_posterior: dict[str, Any],
                              coherence_score: float) -> dict[str, float]:
        """Quantify epistemic uncertainty"""
        # Entropy of mechanism type distribution
        mech_probs = list(mechanism_type_posterior.values())
        if mech_probs:
            mech_entropy = -sum(p * np.log(p + 1e-10) for p in mech_probs if p > 0)
            max_entropy = np.log(len(mech_probs))
            mech_uncertainty = mech_entropy / max_entropy if max_entropy > 0 else get_para
meter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._quantify_unce
rtainty").get("auto_param_L3419_82", 1.0)
        else:
            mech_uncertainty = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ba
yesianMechanismInference._quantify_uncertainty").get("mech_uncertainty", 1.0) # Refactored

        # Sequence completeness uncertainty
        seq_completeness = sequence_posterior.get('sequence_completeness', get_parameter_l
oader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._quantify_uncertainty
").get("auto_param_L3424_75", 0.0))
        seq_uncertainty = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesia
nMechanismInference._quantify_uncertainty").get("auto_param_L3425_26", 1.0) -
seq_completeness

        # Coherence uncertainty
        coherence_uncertainty = get_parameter_loader().get("saaaaaa.analysis.derek_beach.B
ayesianMechanismInference._quantify_uncertainty").get("auto_param_L3428_32", 1.0) -
coherence_score

        # Combined uncertainty
```

```
        total_uncertainty = (
            mech_uncertainty * get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.BayesianMechanismInference._quantify_uncertainty").get("auto_param_L3432_35", 0.4) +
            seq_uncertainty * get_parameter_loader().get("saaaaaa.analysis.derek_beach
.BayesianMechanismInference._quantify_uncertainty").get("auto_param_L3433_34", 0.3) +
            coherence_uncertainty * get_parameter_loader().get("saaaaaa.analysis.derek
_beach.BayesianMechanismInference._quantify_uncertainty").get("auto_param_L3434_40", 0.3)
        )

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


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

        # High total uncertainty
        if uncertainty['total'] > get_parameter_loader().get("saaaaaa.analysis.derek_beach
.BayesianMechanismInference._detect_gaps").get("auto_param_L3451_34", 0.6):
            gaps.append({
                'type': 'high_uncertainty',
                'severity': 'high',
                'message': f"Mecanismo para {node.id} tiene alta incertidumbre
({uncertainty['total']:.2f})",
                'suggestion': "Se requiere más documentación sobre el mecanismo causal"
            })

        # Missing entity
        if not observations.get('entity_activity'):
            gaps.append({
                'type': 'missing_entity',
                'severity': 'high',
                'message': f"No se especifica entidad responsable para {node.id}",
                'suggestion': "Especificar qué entidad ejecutará las actividades"
            })

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

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

        return gaps

    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianMechanismInference._aggregate
_bayesian_confidence")
    def _aggregate_bayesian_confidence(self, confidences: list[float]) -> float:
        """
```

```
        Aggregate multiple Bayesian confidence values.

        Args:
            confidences: List of confidence values to aggregate

        Returns:
            Aggregated confidence value
        """
        if not confidences:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechan
ismInference._aggregate_bayesian_confidence").get("auto_param_L3500_19", 0.5)  # Default
neutral confidence
        return float(np.mean(confidences))

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

        Args:
            mechanism_type: Type of mechanism

        Returns:
            Transition probability matrix
        """
        # Get typical sequence for this mechanism type
        sequence = self.mechanism_sequences.get(mechanism_type, ['planificar', 'ejecutar',
 'evaluar'])
        n = len(sequence)

        # Create a simple sequential transition matrix
        matrix = np.zeros((n, n))
        for i in range(n - 1):
            matrix[i, i + 1] = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ba
yesianMechanismInference._build_transition_matrix").get("auto_param_L3521_31", 0.7)  #
High probability of next step
            matrix[i, i] = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anMechanismInference._build_transition_matrix").get("auto_param_L3522_27", 0.2)       #
Some probability of staying in same step
            if i < n - 2:
                matrix[i, i + 2] = get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.BayesianMechanismInference._build_transition_matrix").get("auto_param_L3524_35", 0.1)  #
 Small probability of skipping
        matrix[n - 1, n - 1] = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ba
yesianMechanismInference._build_transition_matrix").get("auto_param_L3525_31", 1.0)  #
Final state is absorbing

        return matrix

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

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

        Returns:
            Prior probability of transition
        """
        # Same type has high probability
        if from_type == to_type:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechan
ismInference._calculate_type_transition_prior").get("auto_param_L3543_19", 0.7)
```

```
        # Related types have medium probability
        related_pairs = [
            ('administrativo', 'politico'),
            ('tecnico', 'financiero'),
            ('financiero', 'administrativo'),
        ]
        if (from_type, to_type) in related_pairs or (to_type, from_type) in related_pairs:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechan
ismInference._calculate_type_transition_prior").get("auto_param_L3552_19", 0.2)

        # Unrelated types have low probability
        return get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismI
nference._calculate_type_transition_prior").get("auto_param_L3555_15", 0.1)

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

        Args:
            observations: Observed features

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

        # Score each mechanism type
        scores = {}
        for mech_type, typical_verbs in self.mechanism_sequences.items():
            score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianMecha
nismInference._classify_mechanism_type").get("score", 0.0) # Refactored
            # Count matching verbs
            for verb in verbs:
                if any(tv in verb.lower() for tv in typical_verbs):
                    score += get_parameter_loader().get("saaaaaa.analysis.derek_beach.Baye
sianMechanismInference._classify_mechanism_type").get("auto_param_L3580_29", 1.0)
            scores[mech_type] = score

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

        # Adjust for political/administrative entities
        for entity in entities:
            entity_lower = entity.lower()
            if any(word in entity_lower for word in ['alcaldía', 'consejo',
'gobernación']):
                scores['politico'] = scores.get('politico', 0) + get_parameter_loader().ge
t("saaaaaa.analysis.derek_beach.BayesianMechanismInference._classify_mechanism_type").get(
"auto_param_L3591_65", 1.0)
            if any(word in entity_lower for word in ['secretaría', 'dirección',
'oficina']):
                scores['administrativo'] = scores.get('administrativo', 0) + get_parameter
_loader().get("saaaaaa.analysis.derek_beach.BayesianMechanismInference._classify_mechanism
_type").get("auto_param_L3593_77", 1.0)

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

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

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

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

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

    @calibrated_method("saaaaaa.analysis.derek_beach.CausalInferenceSetup.classify_goal_dy
namics")
    def classify_goal_dynamics(self, nodes: dict[str, MetaNode]) -> None:
        """Classify dynamics for each goal"""
        for node in nodes.values():
            text_lower = node.text.lower()

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

    @calibrated_method("saaaaaa.analysis.derek_beach.CausalInferenceSetup.assign_probative
_value")
    def assign_probative_value(self, nodes: dict[str, MetaNode]) -> None:
        """Assign probative test types to nodes"""
        # Import INDICATOR_STRUCTURE from financiero_viabilidad_tablas
        try:
            from financiero_viabilidad_tablas import ColombianMunicipalContext
            indicator_structure = ColombianMunicipalContext.INDICATOR_STRUCTURE
        except ImportError:
            indicator_structure = {
                'resultado': ['línea_base', 'meta', 'año_base', 'año_meta', 'fuente',
'responsable'],
                'producto': ['indicador', 'fórmula', 'unidad_medida', 'línea_base',
'meta', 'periodicidad'],
                'gestión': ['eficacia', 'eficiencia', 'economía', 'costo_beneficio']
            }

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

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

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

            # Perfect Hoop Test: Missing any critical requirement = total hypothesis
failure
            # This applies to producto nodes with D3-Q1 indicators
            if node.type == 'producto':
                if has_linea_base and has_meta and has_fuente:
                    # Perfect indicators trigger Hoop Test classification
                    node.test_type = 'hoop_test'
                    self.logger.debug(f"Meta {node.id} classified as hoop_test (perfect
D3-Q1 compliance)")
```

```
            elif not has_linea_base or not has_meta:
                # Missing critical requirements - still Hoop Test but will fail
                node.test_type = 'hoop_test'
                node.audit_flags.append('hoop_test_failure')
                self.logger.warning(f"Meta {node.id} FAILS hoop_test (missing D3-Q1
critical fields)")
            else:
                node.test_type = 'straw_in_wind'
        # Check for administrative/regulatory nature (Hoop Test)
        elif any(keyword in text_lower for keyword in self.admin_keywords):
            node.test_type = 'hoop_test'
        # Check for highly specific outcomes (Smoking Gun)
        elif node.type == 'resultado' and node.target and node.baseline:
            try:
                float(str(node.target).replace(',', '').replace('%', ''))
                # Smoking Gun: rare, highly specific evidence with strong inferential
power
                node.test_type = 'smoking_gun'
            except (ValueError, TypeError):
                node.test_type = 'straw_in_wind'
        # Double decisive for critical impact goals
        elif node.type == 'impacto' and node.rigor_status == 'fuerte':
            node.test_type = 'doubly_decisive'
        else:
            node.test_type = 'straw_in_wind'

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

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

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

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

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

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

        # HARMONIC FRONT 3 - Enhancement 2: Expand contextual factors
        # Add specific rubric factors for D6-Q5 compliance
        extended_contextual_factors = list(self.contextual_factors) + [
            'restricciones territoriales',
            'restricción territorial',
            'limitación territorial',
            'patrones culturales machistas',
            'machismo',
            'inequidad de género',
            'violencia de género',
            'limitación normativa',
            'limitación legal',
```

```
            'restricción legal',
            'barrera institucional',
            'restricción presupuestal',
            'ausencia de capacidad técnica',
            'baja capacidad institucional',
            'conflicto armado',
            'desplazamiento forzado',
            'población dispersa',
            'ruralidad dispersa',
            'acceso vial limitado',
            'conectividad deficiente'
        ]

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

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

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

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

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

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

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

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

        return failure_points

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

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

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

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

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


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

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

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

            if rigor == 'débil' or not financial:
                color = 'lightcoral'  # Red
            elif audit_flags:
                color = 'lightyellow'  # Yellow
            else:
                color = 'lightgreen'  # Green

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

            entity = node_data.get('responsible_entity')
            if entity:
                label += f"\\n👤 {entity[:30]}"

            if financial:
                label += f"\\n💰 ${financial:,.0f}"
```

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

        # Add edges with causal logic
        for source, target in graph.edges():
            edge_data = graph.edges[source, target]
            keyword = edge_data.get('keyword', '')
            strength = edge_data.get('strength', get_parameter_loader().get("saaaaaa.analy
sis.derek_beach.ReportingEngine.generate_causal_diagram").get("auto_param_L3870_49", 0.5))

            # Determine edge style based on strength
            style = 'solid' if strength > get_parameter_loader().get("saaaaaa.analysis.der
ek_beach.ReportingEngine.generate_causal_diagram").get("auto_param_L3873_42", 0.7) else
'dashed'

            dot_edge = Edge(
                source,
                target,
                label=keyword[:20],
                style=style
            )
            dot.add_edge(dot_edge)

        # Save files
        # Delegate to factory for I/O operation
        from .factory import write_text_file

        dot_path = self.output_dir / f"{policy_code}_causal_diagram.dot"
        png_path = self.output_dir / f"{policy_code}_causal_diagram.png"

        try:
            write_text_file(dot.to_string(), dot_path)
            self.logger.info(f"Diagrama DOT guardado en: {dot_path}")

            # Try to render PNG
            try:
                dot.write_png(str(png_path))
                self.logger.info(f"Diagrama PNG renderizado en: {png_path}")
            except Exception as e:
                self.logger.warning(f"No se pudo renderizar PNG (¿Graphviz instalado?):
{e}")
        except Exception as e:
            self.logger.error(f"Error guardando diagrama: {e}")

        return png_path

    @calibrated_method("saaaaaa.analysis.derek_beach.ReportingEngine.generate_accountabili
ty_matrix")
    def generate_accountability_matrix(self, graph: nx.DiGraph,
                                       policy_code: str) -> Path:
        """Generate accountability matrix in Markdown"""
        md_path = self.output_dir / f"{policy_code}_accountability_matrix.md"

        # Group by impact goals
        impact_goals = [n for n in graph.nodes()
                        if graph.nodes[n].get('type') == 'impacto']

        content = [f"# Matriz de Responsabilidades - {policy_code}\n"]
        content.append("*Generado automáticamente por CDAF v2.0*\n")
        content.append("---\n\n")

        for impact in impact_goals:
            impact_data = graph.nodes[impact]
```

```
            content.append(f"## Meta de Impacto: {impact}\n")
            content.append(f"**Descripción:** {impact_data.get('text', 'N/A')}\n\n")

            # Find all predecessor chains
            predecessors = list(nx.ancestors(graph, impact))

            if predecessors:
                content.append("| Meta | Tipo | Entidad Responsable | Actividad Clave |
Presupuesto |\n")

content.append("|------|------|---------------------|-----------------|-------------|\n")

                for pred in predecessors:
                    pred_data = graph.nodes[pred]
                    meta_type = pred_data.get('type', 'N/A')
                    entity = pred_data.get('responsible_entity', 'No asignado')

                    ea = pred_data.get('entity_activity')
                    activity = 'N/A'
                    if ea and isinstance(ea, dict):
                        activity = ea.get('activity', 'N/A')

                    budget = pred_data.get('financial_allocation')
                    budget_str = f"${budget:,.0f}" if budget else "Sin presupuesto"

                    content.append(f"| {pred} | {meta_type} | {entity} | {activity} |
{budget_str} |\n")

                content.append("\n")
            else:
                content.append("*No se encontraron metas intermedias.*\n\n")

        content.append("\n---\n")
        content.append("### Leyenda\n")
        content.append("- **Meta de Impacto:** Resultado final esperado\n")
        content.append("- **Meta de Resultado:** Cambio intermedio observable\n")
        content.append("- **Meta de Producto:** Entrega tangible del programa\n")

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

        try:
            write_text_file(''.join(content), md_path)
            self.logger.info(f"Matriz de responsabilidades guardada en: {md_path}")
        except Exception as e:
            self.logger.error(f"Error guardando matriz de responsabilidades: {e}")

        return md_path

    @calibrated_method("saaaaaa.analysis.derek_beach.ReportingEngine.generate_confidence_r
eport")
    def generate_confidence_report(self,
                                   nodes: dict[str, MetaNode],
                                   graph: nx.DiGraph,
                                   causal_chains: list[CausalLink],
                                   audit_results: dict[str, AuditResult],
                                   financial_auditor: FinancialAuditor,
                                   sequence_warnings: list[str],
                                   policy_code: str) -> Path:
        """Generate extraction confidence report"""
        json_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"

        # Calculate metrics
        total_metas = len(nodes)

        metas_with_ea = sum(1 for n in nodes.values() if n.entity_activity)
        metas_with_ea_pct = (metas_with_ea / total_metas * 100) if total_metas > 0 else 0
```

```
        enlaces_with_logic = sum(1 for link in causal_chains if link.get('logic'))
        total_edges = graph.number_of_edges()
        enlaces_with_logic_pct = (enlaces_with_logic / total_edges * 100) if total_edges >
 0 else 0

        metas_passed_audit = sum(1 for r in audit_results.values() if r['passed'])
        metas_with_traceability_pct = (metas_passed_audit / total_metas * 100) if
total_metas > 0 else 0

        metas_with_financial = sum(1 for n in nodes.values() if n.financial_allocation)
        metas_with_financial_pct = (metas_with_financial / total_metas * 100) if
total_metas > 0 else 0

        # Node type distribution
        type_distribution = defaultdict(int)
        for node in nodes.values():
            type_distribution[node.type] += 1

        # Rigor distribution
        rigor_distribution = defaultdict(int)
        for node in nodes.values():
            rigor_distribution[node.rigor_status] += 1

        report = {
            "metadata": {
                "policy_code": policy_code,
                "framework_version": "2.get_parameter_loader().get("saaaaaa.analysis.derek
_beach.ReportingEngine.generate_confidence_report").get("auto_param_L4008_40", 0.0)",
                "total_nodes": total_metas,
                "total_edges": total_edges
            },
            "extraction_metrics": {
                "total_metas_identificadas": total_metas,
                "metas_con_EA_extraido": metas_with_ea,
                "metas_con_EA_extraido_pct": round(metas_with_ea_pct, 2),
                "enlaces_con_logica_causal": enlaces_with_logic,
                "enlaces_con_logica_causal_pct": round(enlaces_with_logic_pct, 2),
                "metas_con_trazabilidad_evidencia": metas_passed_audit,
                "metas_con_trazabilidad_evidencia_pct": round(metas_with_traceability_pct,
 2),
                "metas_con_trazabilidad_financiera": metas_with_financial,
                "metas_con_trazabilidad_financiera_pct": round(metas_with_financial_pct,
2)
            },
            "financial_audit": {
                "tablas_financieras_parseadas_exitosamente":
financial_auditor.successful_parses,
                "tablas_financieras_fallidas": financial_auditor.failed_parses,
                "asignaciones_presupuestarias_rastreadas":
len(financial_auditor.financial_data)
            },
            "sequence_audit": {
                "alertas_secuencia_logica": len(sequence_warnings),
                "detalles": sequence_warnings
            },
            "type_distribution": dict(type_distribution),
            "rigor_distribution": dict(rigor_distribution),
            "audit_summary": {
                "total_audited": len(audit_results),
                "passed": sum(1 for r in audit_results.values() if r['passed']),
                "failed": sum(1 for r in audit_results.values() if not r['passed']),
                "total_warnings": sum(len(r['warnings']) for r in audit_results.values()),
                "total_errors": sum(len(r['errors']) for r in audit_results.values())
            },
            "quality_score": self._calculate_quality_score(
                metas_with_traceability_pct,
                metas_with_financial_pct,
                enlaces_with_logic_pct,
```

```
            metas_with_ea_pct
        )
    }

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

    try:
        save_json(report, json_path)
        self.logger.info(f"Reporte de confianza guardado en: {json_path}")
    except Exception as e:
        self.logger.error(f"Error guardando reporte de confianza: {e}")

    return json_path

@calibrated_method("saaaaaa.analysis.derek_beach.ReportingEngine._calculate_quality_sc
ore")
def _calculate_quality_score(self, traceability: float, financial: float,
                             logic: float, ea: float) -> float:
    """Calculate overall quality score (0-100)"""
    weights = {'traceability': get_parameter_loader().get("saaaaaa.analysis.derek_beac
h.ReportingEngine._calculate_quality_score").get("auto_param_L4064_35", 0.35),
'financial': get_parameter_loader().get("saaaaaa.analysis.derek_beach.ReportingEngine._cal
culate_quality_score").get("auto_param_L4064_54", 0.25), 'logic': get_parameter_loader().g
et("saaaaaa.analysis.derek_beach.ReportingEngine._calculate_quality_score").get("auto_para
m_L4064_69", 0.25), 'ea': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Reporti
ngEngine._calculate_quality_score").get("auto_param_L4064_81", 0.15)}
    score = (traceability * weights['traceability'] +
             financial * weights['financial'] +
             logic * weights['logic'] +
             ea * weights['ea'])
    return round(score, 2)

@calibrated_method("saaaaaa.analysis.derek_beach.ReportingEngine.generate_causal_model
_json")
def generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
                               policy_code: str) -> Path:
    """Generate structured JSON export of causal model"""
    json_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"

    # Prepare node data
    nodes_data = {}
    for node_id, node in nodes.items():
        node_dict = asdict(node)
        # Convert NamedTuple to dict
        if node.entity_activity:
            node_dict['entity_activity'] = node.entity_activity._asdict()
        nodes_data[node_id] = node_dict

    # Prepare edge data
    edges_data = []
    for source, target in graph.edges():
        edge_dict = {
            'source': source,
            'target': target,
            **graph.edges[source, target]
        }
        edges_data.append(edge_dict)

    model_data = {
        "policy_code": policy_code,
        "framework_version": "2.get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.ReportingEngine.generate_causal_model_json").get("auto_param_L4098_36", 0.0)",
        "nodes": nodes_data,
        "edges": edges_data,
        "statistics": {
            "total_nodes": len(nodes_data),
            "total_edges": len(edges_data),
```

```
            "node_types": {
                node_type: sum(1 for n in nodes.values() if n.type == node_type)
                for node_type in ['programa', 'producto', 'resultado', 'impacto']
            }
        }
    }

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

    try:
        save_json(model_data, json_path)
        self.logger.info(f"Modelo causal JSON guardado en: {json_path}")
    except Exception as e:
        self.logger.error(f"Error guardando modelo causal: {e}")

    return json_path

class CDAFFramework:
    """Main orchestrator for the CDAF pipeline"""

    def __init__(self, config_path: Path, output_dir: Path, log_level: str = "INFO") ->
None:
    self.logger = logging.getLogger(self.__class__.__name__)
    self.logger.setLevel(getattr(logging, log_level.upper()))

    # Initialize components
    self.config = ConfigLoader(config_path)
    self.output_dir = output_dir

    # Initialize retry handler for external dependencies
    try:
        from retry_handler import DependencyType, get_retry_handler
        self.retry_handler = get_retry_handler()
        retry_enabled = True
    except ImportError:
        self.logger.warning("RetryHandler no disponible, funcionando sin retry logic")
        self.retry_handler = None
        retry_enabled = False

    # Load spaCy model with retry logic
    # Delegate to factory for I/O operation
    from .factory import load_spacy_model

    if retry_enabled and self.retry_handler:
        @self.retry_handler.with_retry(
            DependencyType.SPACY_MODEL,
            operation_name="load_spacy_model",
            exceptions=(OSError, IOError, ImportError)
        )
        def load_spacy_with_retry():
            try:
                nlp = load_spacy_model("es_core_news_lg")
                self.logger.info("Modelo spaCy cargado: es_core_news_lg")
                return nlp
            except OSError:
                self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando
es_core_news_sm...")
                nlp = load_spacy_model("es_core_news_sm")
                return nlp

        try:
            self.nlp = load_spacy_with_retry()
        except OSError:
            self.logger.error("No se encontró ningún modelo de spaCy en español. "
                              "Ejecute: python -m spacy download es_core_news_lg")
            sys.exit(1)
    else:
```

```
            # Fallback to original logic without retry
            try:
                self.nlp = load_spacy_model("es_core_news_lg")
                self.logger.info("Modelo spaCy cargado: es_core_news_lg")
            except OSError:
                self.logger.warning("Modelo es_core_news_lg no encontrado. Intentando
es_core_news_sm...")
                try:
                    self.nlp = load_spacy_model("es_core_news_sm")
                except OSError:
                    self.logger.error("No se encontró ningún modelo de spaCy en español. "
                                      "Ejecute: python -m spacy download es_core_news_lg")
                    sys.exit(1)

        # Initialize modules (pass retry_handler to PDF processor)
        self.pdf_processor = PDFProcessor(self.config, retry_handler=self.retry_handler if
 retry_enabled else None)
        self.causal_extractor = CausalExtractor(self.config, self.nlp)
        self.mechanism_extractor = MechanismPartExtractor(self.config, self.nlp)
        self.bayesian_mechanism = BayesianMechanismInference(self.config, self.nlp)
        self.financial_auditor = FinancialAuditor(self.config)
        self.op_auditor = OperationalizationAuditor(self.config)
        self.inference_setup = CausalInferenceSetup(self.config)
        self.reporting_engine = ReportingEngine(self.config, output_dir)

        # Initialize DNP validator if available
        self.dnp_validator = None
        if DNP_AVAILABLE:
            self.dnp_validator = ValidadorDNP(es_municipio_pdet=False)  # Can be
configured
            self.logger.info("Validador DNP inicializado")

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework.process_document")
    def process_document(self, pdf_path: Path, policy_code: str) -> bool:
        """Main processing pipeline"""
        self.logger.info(f"Iniciando procesamiento de documento: {pdf_path}")

        try:
            # Step 1: Load and extract PDF
            if not self.pdf_processor.load_document(pdf_path):
                return False

            text = self.pdf_processor.extract_text()
            tables = self.pdf_processor.extract_tables()
            self.pdf_processor.extract_sections()

            # Step 2: Extract causal hierarchy
            self.logger.info("Extrayendo jerarquía causal...")
            graph = self.causal_extractor.extract_causal_hierarchy(text)
            nodes = self.causal_extractor.nodes

            # Step 3: Extract Entity-Activity pairs
            self.logger.info("Extrayendo tuplas Entidad-Actividad...")
            for node in nodes.values():
                if node.type == 'producto':
                    ea = self.mechanism_extractor.extract_entity_activity(node.text)
                    if ea:
                        node.entity_activity = ea
                        graph.nodes[node.id]['entity_activity'] = ea._asdict()

            # Step 4: Financial traceability
            self.logger.info("Auditando trazabilidad financiera...")
            self.financial_auditor.trace_financial_allocation(tables, nodes, graph)

            # Step 4.5: Bayesian Mechanism Inference (AGUJA II)
            self.logger.info("Infiriendo mecanismos causales con modelo Bayesiano...")
            inferred_mechanisms = self.bayesian_mechanism.infer_mechanisms(nodes, text)
```

```
        # Step 5: Operationalization audit
        self.logger.info("Auditando operacionalización...")
        audit_results = self.op_auditor.audit_evidence_traceability(nodes)
        sequence_warnings = self.op_auditor.audit_sequence_logic(graph)

        # Step 5.5: Bayesian Counterfactual Audit (AGUJA III)
        # Note: pdet_alignment should be calculated separately if needed via
financiero_viabilidad_tablas
        # For now, using None as placeholder - can be enhanced by integrating
PDETMunicipalPlanAnalyzer
        self.logger.info("Ejecutando auditoría contrafactual Bayesiana...")
        counterfactual_audit = self.op_auditor.bayesian_counterfactual_audit(nodes,
graph, pdet_alignment=None)

        # Step 6: Causal inference setup
        self.logger.info("Preparando para inferencia causal...")
        self.inference_setup.classify_goal_dynamics(nodes)
        self.inference_setup.assign_probative_value(nodes)
        self.inference_setup.identify_failure_points(graph, text)

        # Step 7: DNP Standards Validation (if available)
        if self.dnp_validator:
            self.logger.info("Validando cumplimiento de estándares DNP...")
            self._validate_dnp_compliance(nodes, graph, policy_code)

        # Step 8: Generate reports
        self.logger.info("Generando reportes y visualizaciones...")
        self.reporting_engine.generate_causal_diagram(graph, policy_code)
        self.reporting_engine.generate_accountability_matrix(graph, policy_code)
        self.reporting_engine.generate_confidence_report(
            nodes, graph, self.causal_extractor.causal_chains,
            audit_results, self.financial_auditor, sequence_warnings, policy_code
        )
        self.reporting_engine.generate_causal_model_json(graph, nodes, policy_code)

        # Step 8: Generate Bayesian inference reports
        self.logger.info("Generando reportes de inferencia Bayesiana...")
        self._generate_bayesian_reports(
            inferred_mechanisms, counterfactual_audit, policy_code
        )

        # Step 9: Self-reflective learning from audit results (frontier paradigm)
        if self.config.validated_config and
self.config.validated_config.self_reflection.enable_prior_learning:
            self.logger.info("Actualizando priors con retroalimentación del
análisis...")
            feedback_data = self._extract_feedback_from_audit(
                inferred_mechanisms, counterfactual_audit, audit_results
            )
            self.config.update_priors_from_feedback(feedback_data)

            # HARMONIC FRONT 4: Check uncertainty reduction criterion
            if hasattr(self.bayesian_mechanism, '_mean_mechanism_uncertainty'):
                uncertainty_check = self.config.check_uncertainty_reduction_criterion(
                    self.bayesian_mechanism._mean_mechanism_uncertainty
                )
                self.logger.info(
                    f"Uncertainty criterion check: {uncertainty_check['status']} "
                    f"({uncertainty_check['iterations_tracked']}/10 iterations, "
                    f"{uncertainty_check['reduction_percent']:.2f}% reduction)"
                )

        self.logger.info(f"✓ Procesamiento completado exitosamente para
{policy_code}")
        return True

    except CDAFException as e:
        # Structured error handling with custom exceptions
```

```
            self.logger.error(f"Error CDAF: {e.message}")
            self.logger.error(f"Detalles: {json.dumps(e.to_dict(), indent=2)}")
            if not e.recoverable:
                raise
            return False
        except Exception as e:
            # Wrap unexpected errors in CDAFProcessingError
            raise CDAFProcessingError(
                "Error crítico en el procesamiento",
                details={'error': str(e), 'type': type(e).__name__},
                stage="document_processing",
                recoverable=False
            ) from e

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._extract_feedback_from_
audit")
    def _extract_feedback_from_audit(self, inferred_mechanisms: dict[str, dict[str, Any]],
                                     counterfactual_audit: dict[str, Any],
                                     audit_results: dict[str, AuditResult]) -> dict[str,
Any]:
        """
        Extract feedback data from audit results for self-reflective prior updating

        This implements the frontier paradigm of learning from audit results
        to improve future inference accuracy.

        HARMONIC FRONT 4 ENHANCEMENT:
        - Reduces mechanism_type_priors for mechanisms with implementation_failure flags
        - Tracks necessity/sufficiency test failures
        - Penalizes "miracle" mechanisms that fail counterfactual tests
        """
        feedback = {}

        # Extract mechanism type frequencies from successful inferences
        mechanism_frequencies = defaultdict(float)
        failure_frequencies = defaultdict(float)  # NEW: Track failures
        total_mechanisms = 0
        total_failures = 0

        # Get causal implications from audit
        causal_implications = counterfactual_audit.get('causal_implications', {})

        for node_id, mechanism in inferred_mechanisms.items():
            mechanism_type_dist = mechanism.get('mechanism_type', {})
            # Weight by confidence (coherence score)
            confidence = mechanism.get('coherence_score', get_parameter_loader().get("saaa
aaa.analysis.derek_beach.CDAFFramework._extract_feedback_from_audit").get("auto_param_L434
0_58", 0.5))

            # Check for implementation_failure flags in audit results
            node_implications = causal_implications.get(node_id, {})
            causal_effects = node_implications.get('causal_effects', {})
            has_implementation_failure = 'implementation_failure' in causal_effects

            # Check necessity/sufficiency test results
            necessity_test = mechanism.get('necessity_test', {})
            sufficiency_test = mechanism.get('sufficiency_test', {})
            failed_necessity = not necessity_test.get('is_necessary', True)
            failed_sufficiency = not sufficiency_test.get('is_sufficient', True)

            # If mechanism failed tests or has implementation_failure flag
            if has_implementation_failure or failed_necessity or failed_sufficiency:
                total_failures += 1
                # Track which mechanism types are associated with failures
                for mech_type, prob in mechanism_type_dist.items():
                    failure_frequencies[mech_type] += prob * confidence
            else:
                # Only count successes for positive reinforcement
```

```
                for mech_type, prob in mechanism_type_dist.items():
                    mechanism_frequencies[mech_type] += prob * confidence
                    total_mechanisms += confidence

        # Normalize frequencies
        if total_mechanisms > 0:
            mechanism_frequencies = {
                k: v / total_mechanisms
                for k, v in mechanism_frequencies.items()
            }
            feedback['mechanism_frequencies'] = dict(mechanism_frequencies)

        # NEW: Calculate penalty factors for failed mechanism types
        if total_failures > 0:
            failure_frequencies = {
                k: v / total_failures
                for k, v in failure_frequencies.items()
            }
            feedback['failure_frequencies'] = dict(failure_frequencies)

            # Calculate penalty: reduce priors for frequently failing types
            penalty_factors = {}
            for mech_type, failure_freq in failure_frequencies.items():
                # Higher failure frequency = stronger penalty (get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.CDAFFramework._extract_feedback_from_audit").get("auto_param
_L4384_63", 0.7) to get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework
._extract_feedback_from_audit").get("auto_param_L4384_70", 0.95) reduction)
                penalty_factors[mech_type] = get_parameter_loader().get("saaaaaa.analysis.
derek_beach.CDAFFramework._extract_feedback_from_audit").get("auto_param_L4385_45", 0.95)
- (failure_freq * get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework._
extract_feedback_from_audit").get("auto_param_L4385_68", 0.25))
            feedback['penalty_factors'] = penalty_factors

        # Add audit quality metrics for future reference
        feedback['audit_quality'] = {
            'total_nodes_audited': len(audit_results),
            'passed_count': sum(1 for r in audit_results.values() if r['passed']),
            'success_rate': sum(1 for r in audit_results.values() if r['passed']) /
max(len(audit_results), 1),
            'failure_count': total_failures,  # NEW
            'failure_rate': total_failures / max(len(inferred_mechanisms), 1)  # NEW
        }

        # Track necessity/sufficiency failures for iterative validation loop
        necessity_failures = sum(1 for m in inferred_mechanisms.values()
                                 if not m.get('necessity_test', {}).get('is_necessary',
True))
        sufficiency_failures = sum(1 for m in inferred_mechanisms.values()
                                   if not m.get('sufficiency_test',
{}).get('is_sufficient', True))

        feedback['test_failures'] = {
            'necessity_failures': necessity_failures,
            'sufficiency_failures': sufficiency_failures
        }

        return feedback


@calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._validate_dnp_compliance")
    def _validate_dnp_compliance(self, nodes: dict[str, MetaNode],
                                 graph: nx.DiGraph, policy_code: str) -> None:
        """
        Validate DNP compliance for all nodes/projects
        Generates DNP compliance report
        """
        if not self.dnp_validator:
            return
```

```
        # Build project list from nodes
        proyectos = []
        for node_id, node in nodes.items():
            # Extract sector from responsible entity or type
            sector = "general"
            if node.responsible_entity:
                entity_lower = node.responsible_entity.lower()
                if "educaci" in entity_lower or "edu" in entity_lower:
                    sector = "educacion"
                elif "salud" in entity_lower:
                    sector = "salud"
                elif "agua" in entity_lower or "acueducto" in entity_lower:
                    sector = "agua_potable_saneamiento"
                elif (
                    "via" in entity_lower or "vial" in entity_lower or "transporte" in
 entity_lower or "infraestructura" in entity_lower):
                    sector = "vias_transporte"
                elif "agr" in entity_lower or "rural" in entity_lower:
                    sector = "desarrollo_agropecuario"

            # Infer indicators from node type
            indicadores = []
            if node.type == "producto":
                # Map to MGA product indicators based on sector
                if sector == "educacion":
                    indicadores = ["EDU-020", "EDU-021"]
                elif sector == "salud":
                    indicadores = ["SAL-020", "SAL-021"]
                elif sector == "agua_potable_saneamiento":
                    indicadores = ["APS-020", "APS-021"]
            elif node.type == "resultado":
                # Map to MGA result indicators
                if sector == "educacion":
                    indicadores = ["EDU-001", "EDU-002"]
                elif sector == "salud":
                    indicadores = ["SAL-001", "SAL-002"]
                elif sector == "agua_potable_saneamiento":
                    indicadores = ["APS-001", "APS-002"]

            proyectos.append({
                "nombre": node_id,
                "sector": sector,
                "descripcion": node.text[:200] if node.text else "",
                "indicadores": indicadores,
                "presupuesto": node.financial_allocation or get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.CDAFFramework._validate_dnp_compliance").get("auto_param_L4463_
60", 0.0),
                "es_rural": "rural" in node.text.lower() if node.text else False,
                "poblacion_victimas": "v ctima" in node.text.lower() if node.text else
False
            })

        # Validate each project
        dnp_results = []
        for proyecto in proyectos:
            resultado = self.dnp_validator.validar_proyecto_integral(
                sector=proyecto["sector"],
                descripcion=proyecto["descripcion"],
                indicadores_propuestos=proyecto["indicadores"],
                presupuesto=proyecto["presupuesto"],
                es_rural=proyecto["es_rural"],
                poblacion_victimas=proyecto["poblacion_victimas"]
            )
            dnp_results.append({
                "proyecto": proyecto["nombre"],
                "resultado": resultado
            })
```

```
        # Generate DNP compliance report
        self._generate_dnp_report(dnp_results, policy_code)

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._generate_dnp_report")
    def _generate_dnp_report(self, dnp_results: list[dict], policy_code: str) -> None:
        """Generate comprehensive DNP compliance report"""
        report_path = self.output_dir / f"{policy_code}{DNP_REPORT_SUFFIX}"

        total_proyectos = len(dnp_results)
        if total_proyectos == 0:
            return

        # Calculate aggregate statistics
        proyectos_excelente = sum(1 for r in dnp_results
                                  if r["resultado"].nivel_cumplimiento.value ==
"excelente")
        proyectos_bueno = sum(1 for r in dnp_results
                              if r["resultado"].nivel_cumplimiento.value == "bueno")
        proyectos_aceptable = sum(1 for r in dnp_results
                                  if r["resultado"].nivel_cumplimiento.value ==
"aceptable")
        proyectos_insuficiente = sum(1 for r in dnp_results
                                     if r["resultado"].nivel_cumplimiento.value ==
"insuficiente")

        score_promedio = sum(r["resultado"].score_total for r in dnp_results) /
total_proyectos

        # Build report
        lines = []
        lines.append("=" * 100)
        lines.append("REPORTE DE CUMPLIMIENTO DE ESTÁNDARES DNP")
        lines.append(f"Código de Política: {policy_code}")
        lines.append("=" * 100)
        lines.append("")

        lines.append("RESUMEN EJECUTIVO")
        lines.append("-" * 100)
        lines.append(f"Total de Proyectos/Metas Analizados: {total_proyectos}")
        lines.append(f"Score Promedio de Cumplimiento: {score_promedio:.1f}/100")
        lines.append("")
        lines.append("Distribución por Nivel de Cumplimiento:")
        lines.append(
            f"  • Excelente (>90%):      {proyectos_excelente:3d} ({proyectos_excelente /
total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  • Bueno (75-90%):        {proyectos_bueno:3d} ({proyectos_bueno /
total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  • Aceptable (60-75%):    {proyectos_aceptable:3d} ({proyectos_aceptable /
total_proyectos * 100:5.1f}%)")
        lines.append(
            f"  • Insuficiente (<60%):   {proyectos_insuficiente:3d}
({proyectos_insuficiente / total_proyectos * 100:5.1f}%)")
        lines.append("")

        # Detailed validation per project
        lines.append("VALIDACIÓN DETALLADA POR PROYECTO/META")
        lines.append("=" * 100)

        for i, result_data in enumerate(dnp_results, 1):
            proyecto = result_data["proyecto"]
            resultado = result_data["resultado"]

            lines.append("")
            lines.append(f"{i}. {proyecto}")
            lines.append("-" * 100)
```

```
            lines.append(
                f"   Score: {resultado.score_total:.1f}/100 | Nivel:
{resultado.nivel_cumplimiento.value.upper()}")

            # Competencies
            comp_status = "✓" if resultado.cumple_competencias else "✗"
            lines.append(f"   Competencias Municipales: {comp_status}")
            if resultado.competencias_validadas:
                lines.append(f"     - Aplicables: {',
'.join(resultado.competencias_validadas[:3])}")

            # MGA Indicators
            mga_status = "✓" if resultado.cumple_mga else "✗"
            lines.append(f"   Indicadores MGA: {mga_status}")
            if resultado.indicadores_mga_usados:
                lines.append(f"     - Usados: {',
'.join(resultado.indicadores_mga_usados)}")
            if resultado.indicadores_mga_faltantes:
                lines.append(f"     - Recomendados: {',
'.join(resultado.indicadores_mga_faltantes)}")

            # PDET (if applicable)
            if resultado.es_municipio_pdet:
                pdet_status = "✓" if resultado.cumple_pdet else "✗"
                lines.append(f"   Lineamientos PDET: {pdet_status}")
                if resultado.lineamientos_pdet_cumplidos:
                    lines.append(f"     - Cumplidos:
{len(resultado.lineamientos_pdet_cumplidos)}")

            # Critical alerts
            if resultado.alertas_criticas:
                lines.append("   ⚠ ALERTAS CRÍTICAS:")
                for alerta in resultado.alertas_criticas:
                    lines.append(f"     - {alerta}")

            # Recommendations
            if resultado.recomendaciones:
                lines.append("   💡 RECOMENDACIONES:")
                for rec in resultado.recomendaciones[:3]:  # Top 3
                    lines.append(f"     - {rec}")

        lines.append("")
        lines.append("=" * 100)
        lines.append("NORMATIVA DE REFERENCIA")
        lines.append("-" * 100)
        lines.append("• Competencias Municipales: Ley 136/1994, Ley 715/2001, Ley
1551/2012")
        lines.append("• Indicadores MGA: DNP - Metodología General Ajustada")
        lines.append("• PDET: Decreto 893/2017, Acuerdo Final de Paz")
        lines.append("=" * 100)

        # Write report
        # Delegate to factory for I/O operation
        from .factory import write_text_file

        try:
            write_text_file('\n'.join(lines), report_path)
            self.logger.info(f"Reporte de cumplimiento DNP guardado en: {report_path}")
        except Exception as e:
            self.logger.error(f"Error guardando reporte DNP: {e}")


@calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._audit_causal_coherence")
    def _audit_causal_coherence(self, graph: nx.DiGraph, nodes: dict[str, MetaNode]) ->
dict[str, Any]:
        """
        Audit causal coherence of the extracted model.
```

```
        Args:
            graph: Causal graph
            nodes: Dictionary of nodes

        Returns:
            Dictionary with coherence audit results
        """
        audit = {
            'total_nodes': len(nodes),
            'total_edges': graph.number_of_edges(),
            'disconnected_nodes': [],
            'cycles': [],
            'coherence_score': get_parameter_loader().get("saaaaaa.analysis.derek_beach.CD
AFFramework._audit_causal_coherence").get("auto_param_L4615_31", 0.0)
        }

        # Check for disconnected nodes
        for node_id in nodes:
            if graph.has_node(node_id) and graph.degree(node_id) == 0:
                audit['disconnected_nodes'].append(node_id)

        # Check for cycles (should not exist in causal DAG)
        try:
            cycles = list(nx.simple_cycles(graph))
            audit['cycles'] = cycles
        except:
            pass

        # Calculate coherence score
        connected_ratio = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFra
mework._audit_causal_coherence").get("auto_param_L4631_26", 1.0) -
(len(audit['disconnected_nodes']) / max(len(nodes), 1))
        acyclic_score = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFrame
work._audit_causal_coherence").get("auto_param_L4632_24", 1.0) if len(audit['cycles']) ==
0 else get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework._audit_causa
l_coherence").get("auto_param_L4632_62", 0.5)
        audit['coherence_score'] = (connected_ratio + acyclic_score) / 2.0

        return audit

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._generate_causal_model_
json")
    def _generate_causal_model_json(self, graph: nx.DiGraph, nodes: dict[str, MetaNode],
                                    policy_code: str) -> None:
        """
        Generate JSON representation of causal model.

        Args:
            graph: Causal graph
            nodes: Dictionary of nodes
            policy_code: Policy code for filename
        """
        model = {
            'policy_code': policy_code,
            'nodes': [],
            'edges': []
        }

        # Add nodes
        for node_id, node in nodes.items():
            model['nodes'].append({
                'id': node_id,
                'text': node.text,
                'type': node.type,
                'baseline': str(node.baseline) if node.baseline else None,
                'target': str(node.target) if node.target else None
            })
```

```
        # Add edges
        for source, target in graph.edges():
            edge_data = graph.get_edge_data(source, target)
            model['edges'].append({
                'source': source,
                'target': target,
                'logic': edge_data.get('logic', 'unknown'),
                'strength': edge_data.get('strength', get_parameter_loader().get("saaaaaa.
analysis.derek_beach.CDAFFramework._generate_causal_model_json").get("auto_param_L4671_54"
, 0.5))
            })

        # Write to file
        output_path = self.output_dir / f"{policy_code}{CAUSAL_MODEL_SUFFIX}"
        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(model, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Causal model JSON saved to: {output_path}")
        except Exception as e:
            self.logger.error(f"Error saving causal model JSON: {e}")

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._generate_dnp_complianc
e_report")
    def _generate_dnp_compliance_report(self, nodes: dict[str, MetaNode],
                                        policy_code: str) -> dict[str, Any]:
        """
        Generate DNP compliance report.

        Args:
            nodes: Dictionary of nodes
            policy_code: Policy code

        Returns:
            Compliance report dictionary
        """
        report = {
            'policy_code': policy_code,
            'total_products': 0,
            'compliant_products': 0,
            'compliance_rate': get_parameter_loader().get("saaaaaa.analysis.derek_beach.CD
AFFramework._generate_dnp_compliance_report").get("auto_param_L4700_31", 0.0),
            'gaps': []
        }

        # Check products for DNP compliance
        for node_id, node in nodes.items():
            if node.type == 'producto':
                report['total_products'] += 1

                # Check required fields
                has_baseline = node.baseline is not None
                has_target = node.target is not None
                has_indicator = len(node.text) > 10  # Simple check

                is_compliant = has_baseline and has_target and has_indicator

                if is_compliant:
                    report['compliant_products'] += 1
                else:
                    gaps = []
                    if not has_baseline:
                        gaps.append('missing_baseline')
                    if not has_target:
                        gaps.append('missing_target')
                    if not has_indicator:
                        gaps.append('missing_indicator')

                    report['gaps'].append({
```

```
                    'node_id': node_id,
                    'issues': gaps
                })

        if report['total_products'] > 0:
            report['compliance_rate'] = report['compliant_products'] /
report['total_products']

        return report

    @calibrated_method("saaaaaa.analysis.derek_beach.CDAFFramework._generate_extraction_re
port")
    def _generate_extraction_report(self, nodes: dict[str, MetaNode],
                                    graph: nx.DiGraph,
                                    policy_code: str) -> None:
        """
        Generate extraction confidence report.

        Args:
            nodes: Dictionary of nodes
            graph: Causal graph
            policy_code: Policy code
        """
        report = {
            'policy_code': policy_code,
            'extraction_summary': {
                'total_nodes': len(nodes),
                'total_edges': graph.number_of_edges(),
                'nodes_by_type': {}
            },
            'node_confidence': []
        }

        # Count nodes by type
        for node in nodes.values():
            node_type = node.type
            report['extraction_summary']['nodes_by_type'][node_type] = \
                report['extraction_summary']['nodes_by_type'].get(node_type, 0) + 1

        # Add confidence scores
        for node_id, node in nodes.items():
            confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFram
ework._generate_extraction_report").get("confidence", 0.8) # Refactored
            if hasattr(node, 'rigor_status'):
                if node.rigor_status == 'fuerte':
                    confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.
CDAFFramework._generate_extraction_report").get("confidence", 0.9) # Refactored
                elif node.rigor_status == 'débil':
                    confidence = get_parameter_loader().get("saaaaaa.analysis.derek_beach.
CDAFFramework._generate_extraction_report").get("confidence", 0.6) # Refactored

            report['node_confidence'].append({
                'node_id': node_id,
                'confidence': confidence
            })

        # Write report
        output_path = self.output_dir / f"{policy_code}{EXTRACTION_REPORT_SUFFIX}"
        try:
            with open(output_path, 'w', encoding='utf-8') as f:
                json.dump(report, f, indent=2, ensure_ascii=False)
            self.logger.info(f"Extraction report saved to: {output_path}")
        except Exception as e:
            self.logger.error(f"Error saving extraction report: {e}")

# ============================================================================
# AGUJA I: PRIOR ADAPTATIVO (EVIDENCIA → BAYES)
# ============================================================================
```

```
class BayesFactorTable:
    """Tabla fija de Bayes Factors por tipo de test evidencial (Beach & Pedersen 2019)"""
    FACTORS = {
        'straw': (get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework._
generate_extraction_report").get("auto_param_L4795_18", 1.0), 1.5),      # STRAW_IN_WIND:
Weak evidence
        'hoop': (3.0, 5.0),       # HOOP TEST: Necessary but not sufficient
        'smoking': (1get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramewor
k._generate_extraction_report").get("auto_param_L4797_21", 0.0), 3get_parameter_loader().g
et("saaaaaa.analysis.derek_beach.CDAFFramework._generate_extraction_report").get("auto_par
am_L4797_27", 0.0)),  # SMOKING GUN: Sufficient but not necessary
        'doubly': (5get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework
._generate_extraction_report").get("auto_param_L4798_20", 0.0), 10get_parameter_loader().g
et("saaaaaa.analysis.derek_beach.CDAFFramework._generate_extraction_report").get("auto_par
am_L4798_27", 0.0))   # DOUBLY DECISIVE: Necessary AND sufficient
    }

    @classmethod
    def get_bayes_factor(cls, test_type: str) -> float:
        """Obtiene BF medio para tipo de test"""
        if test_type not in cls.FACTORS:
            return 1.5  # Default straw-in-wind
        min_bf, max_bf = cls.FACTORS[test_type]
        return (min_bf + max_bf) / 2.0

    @classmethod
    def get_version(cls) -> str:
        """Version de tabla BF para trazabilidad"""
        return "Beach2019_vget_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFr
amework._generate_extraction_report").get("auto_param_L4812_27", 1.0)"

class AdaptivePriorCalculator:
    """
    AGUJA I - Prior Adaptativo con Bayes Factor y calibración

    PROMPT I-1: Ponderación evidencial con BF y calibración
    Mapea test_type→BayesFactor, calcula likelihood adaptativo combinando
    dominios {semantic, temporal, financial, structural} con pesos normalizados.

    PROMPT I-2: Sensibilidad, OOD y ablation evidencial
    Perturba cada componente ±10% y reporta ∂p/∂component top-3.

    PROMPT I-3: Trazabilidad y reproducibilidad
    Con semilla fija, guarda bf_table_version, weights_version, snippets.

    QUALITY CRITERIA:
    - BrierScore ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework.
_generate_extraction_report").get("auto_param_L4829_19", 0.20) en validación sintética
    - ACE ∈ [−get_parameter_loader().get("saaaaaa.analysis.derek_beach.CDAFFramework._gene
rate_extraction_report").get("auto_param_L4830_14", 0.02), get_parameter_loader().get("saa
aaaa.analysis.derek_beach.CDAFFramework._generate_extraction_report").get("auto_param_L483
0_20", 0.02)] (Average Calibration Error)
    - Cobertura CI95% ∈ [92%, 98%]
    - Monotonicidad: ↑ señales → ¬↓ p_mechanism
    """

    def __init__(self, calibration_params: dict[str, float] | None = None) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.bf_table = BayesFactorTable()

        # Calibration params: logit⁻¹(α + β·score)
        self.calibration = calibration_params or {
            'alpha': -2.0,  # Intercept
            'beta': 4.0     # Slope
        }

        # Domain weights (normalized)
```

```
        self.default_domain_weights = {
            'semantic': get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptiveP
riorCalculator.__init__").get("auto_param_L4847_24", 0.35),
            'temporal': get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptiveP
riorCalculator.__init__").get("auto_param_L4848_24", 0.25),
            'financial': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Adaptive
PriorCalculator.__init__").get("auto_param_L4849_25", 0.25),
            'structural': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Adaptiv
ePriorCalculator.__init__").get("auto_param_L4850_26", 0.15)
        }

    def calculate_likelihood_adaptativo(
        self,
        evidence_dict: dict[str, Any],
        test_type: str = 'hoop'
    ) -> dict[str, Any]:
        """
        PROMPT I-1: Calcula likelihood adaptativo con BF y dominios

        Args:
            evidence_dict: Evidencia por caso {semantic, temporal, financial, structural}
            test_type: Tipo de test evidencial (straw, hoop, smoking, doubly)

        Returns:
            Dict con p_mechanism, BF_used, domain_weights, triangulation_bonus, etc.
        """
        # 1. Obtener Bayes Factor para test_type
        bf_used = self.bf_table.get_bayes_factor(test_type)

        # 2. Extraer scores por dominio
        domain_scores = {
            'semantic': evidence_dict.get('semantic', {}).get('score', get_parameter_loade
r().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_param_L
4873_71", 0.0)),
            'temporal': evidence_dict.get('temporal', {}).get('score', get_parameter_loade
r().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_param_L
4874_71", 0.0)),
            'financial': evidence_dict.get('financial', {}).get('score', get_parameter_loa
der().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_param
_L4875_73", 0.0)),
            'structural': evidence_dict.get('structural', {}).get('score', get_parameter_l
oader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_par
am_L4876_75", 0.0))
        }

        # 3. Ajustar pesos si falta dominio (baja peso a 0, reparte)
        adjusted_weights = self._adjust_domain_weights(domain_scores)

        # 4. Calcular score combinado normalizado
        combined_score = sum(
            domain_scores[domain] * adjusted_weights[domain]
            for domain in domain_scores
        )

        # 5. Aplicar multiplicador BF normalizado
        all_bfs = [np.mean(bf_range) for bf_range in self.bf_table.FACTORS.values()]
        mean_bf = np.mean(all_bfs)
        bf_multiplier = bf_used / mean_bf
        adapted_score = combined_score * bf_multiplier

        # 6. Bonus de triangulación si ≥3 dominios activos
        active_domains = sum(1 for s in domain_scores.values() if s > get_parameter_loader
().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_param_L4
895_70", 0.1))
        triangulation_bonus = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ada
ptivePriorCalculator.__init__").get("auto_param_L4896_30", 0.05) if active_domains >= 3
else get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.__in
it__").get("auto_param_L4896_63", 0.0)
```

```
        final_score = min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.Adaptiv
ePriorCalculator.__init__").get("auto_param_L4898_26", 1.0), adapted_score +
triangulation_bonus)

        # 7. Transformar a probabilidad con logit inverso: p = 1/(1+exp(-(α+β·score)))
        alpha = self.calibration['alpha']
        beta = self.calibration['beta']
        logit_value = alpha + beta * final_score
        p_mechanism = get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePri
orCalculator.__init__").get("auto_param_L4904_22", 1.0) / (get_parameter_loader().get("saa
aaaa.analysis.derek_beach.AdaptivePriorCalculator.__init__").get("auto_param_L4904_29",
1.0) + np.exp(-logit_value))

        # 8. Clip [1e-6, 1-1e-6]
        p_mechanism = np.clip(p_mechanism, 1e-6, 1 - 1e-6)

        return {
            'p_mechanism': float(p_mechanism),
            'BF_used': bf_used,
            'domain_weights': adjusted_weights,
            'triangulation_bonus': triangulation_bonus,
            'calibration_params': self.calibration,
            'test_type': test_type,
            'combined_score': combined_score,
            'active_domains': active_domains
        }

    @calibrated_method("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._adjust_domai
n_weights")
    def _adjust_domain_weights(self, domain_scores: dict[str, float]) -> dict[str, float]:
        """Ajusta pesos si falta dominio: baja a 0 y reparte"""
        adjusted = self.default_domain_weights.copy()

        # Identificar dominios faltantes (score ≤ 0)
        missing_domains = [d for d, s in domain_scores.items() if s <= 0]

        if missing_domains:
            # Bajar peso a 0 para dominios faltantes
            total_missing_weight = sum(adjusted[d] for d in missing_domains)
            for d in missing_domains:
                adjusted[d] = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ada
ptivePriorCalculator._adjust_domain_weights").get("auto_param_L4932_30", 0.0)

            # Repartir peso entre dominios activos
            active_domains = [d for d in adjusted if adjusted[d] > 0]
            if active_domains:
                bonus_per_domain = total_missing_weight / len(active_domains)
                for d in active_domains:
                    adjusted[d] += bonus_per_domain

        # Renormalizar para asegurar suma = get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L4941_44",
1.0)
        total = sum(adjusted.values())
        if total > 0:
            adjusted = {k: v / total for k, v in adjusted.items()}

        return adjusted

    def sensitivity_analysis(
        self,
        evidence_dict: dict[str, Any],
        test_type: str = 'hoop',
        perturbation: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ada
ptivePriorCalculator._adjust_domain_weights").get("auto_param_L4952_30", 0.10)
    ) -> dict[str, Any]:
        """
```

PROMPT I-2: Sensibilidad, OOD y ablation evidencial

Perturba cada componente ±10% y reporta ∂p/∂component top-3.
Ejecuta ablaciones: sólo textual, sólo financiero, sólo estructural.

CRITERIA:
- |delta_p_sensitivity|_max ≤ get_parameter_loader().get("saaaaaa.analysis.derek_b
each.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L4961_38", 0.15)
- sign_concordance ≥ 2/3
- OOD_drop ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePrio
rCalculator._adjust_domain_weights").get("auto_param_L4963_21", 0.10)
"""
# Baseline
baseline_result = self.calculate_likelihood_adaptativo(evidence_dict, test_type)
baseline_p = baseline_result['p_mechanism']

# 1. Sensibilidad por componente
sensitivity_map = {}
for domain in ['semantic', 'temporal', 'financial', 'structural']:
    if domain in evidence_dict and isinstance(evidence_dict[domain], dict) and
'score' in evidence_dict[domain]:
        # Perturbar +10%
        perturbed_evidence = self._perturb_evidence(evidence_dict, domain,
perturbation)
        perturbed_result =
self.calculate_likelihood_adaptativo(perturbed_evidence, test_type)
        delta_p = perturbed_result['p_mechanism'] - baseline_p

        sensitivity_map[domain] = {
            'delta_p': delta_p,
            'relative_change': delta_p / max(baseline_p, 1e-6)
        }

# Top-3 por magnitud
top_3 = sorted(
    sensitivity_map.items(),
    key=lambda x: abs(x[1]['delta_p']),
    reverse=True
)[:3]

# 2. Ablaciones: sólo un dominio
ablation_results = {}
for domain in ['semantic', 'financial', 'structural']:
    ablated_evidence = {
        domain: evidence_dict.get(domain, {'score': get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_para
m_L4994_60", 0.0)})
    }
    if ablated_evidence[domain].get('score', 0) > 0:
        abl_result = self.calculate_likelihood_adaptativo(ablated_evidence,
test_type)
        ablation_results[f'only_{domain}'] = {
            'p_mechanism': abl_result['p_mechanism'],
            'sign_match': (abl_result['p_mechanism'] > get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_p
aram_L5000_63", 0.5)) == (baseline_p > get_parameter_loader().get("saaaaaa.analysis.derek_
beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L5000_85", 0.5))
        }

# Sign concordance
sign_concordance = sum(
    1 for r in ablation_results.values() if r['sign_match']
) / max(len(ablation_results), 1)

# 3. OOD con ruido
ood_evidence = self._add_ood_noise(evidence_dict)
ood_result = self.calculate_likelihood_adaptativo(ood_evidence, test_type)
ood_drop = abs(baseline_p - ood_result['p_mechanism'])

```
        # 4. Evaluación de criterios
        max_sensitivity = max((abs(item[1]['delta_p']) for item in top_3), default=get_par
ameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._adjust_domain_w
eights").get("auto_param_L5014_83", 0.0))
        criteria_met = {
            'max_sensitivity_ok': max_sensitivity <= get_parameter_loader().get("saaaaaa.a
nalysis.derek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L5016
_53", 0.15),
            'sign_concordance_ok': sign_concordance >= 2/3,
            'ood_drop_ok': ood_drop <= get_parameter_loader().get("saaaaaa.analysis.derek_
beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L5018_39", 0.10)
        }

        # Determinar si caso es frágil
        is_fragile = not all(criteria_met.values())

        return {
            'influence_top3': [(domain, data['delta_p']) for domain, data in top_3],
            'delta_p_sensitivity': max_sensitivity,
            'sign_concordance': sign_concordance,
            'OOD_drop': ood_drop,
            'ablation_results': ablation_results,
            'criteria_met': criteria_met,
            'is_fragile': is_fragile,
            'recommendation': 'downgrade' if is_fragile else 'accept'
        }

    def _perturb_evidence(
        self,
        evidence_dict: dict[str, Any],
        domain: str,
        perturbation: float
    ) -> dict[str, Any]:
        """Perturba un dominio específico"""
        import copy
        perturbed = copy.deepcopy(evidence_dict)
        if domain in perturbed and isinstance(perturbed[domain], dict) and 'score' in
perturbed[domain]:
            perturbed[domain]['score'] *= (get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L5045_43", 1.0)
 + perturbation)
            perturbed[domain]['score'] = min(get_parameter_loader().get("saaaaaa.analysis.
derek_beach.AdaptivePriorCalculator._adjust_domain_weights").get("auto_param_L5046_45",
1.0), perturbed[domain]['score'])
        return perturbed


@calibrated_method("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._add_ood_noise")
    def _add_ood_noise(self, evidence_dict: dict[str, Any]) -> dict[str, Any]:
        """Genera set OOD con ruido semántico y tablas malformadas"""
        import copy
        ood = copy.deepcopy(evidence_dict)

        # Agregar ruido gaussiano a todos los scores
        for domain in ood:
            if isinstance(ood[domain], dict) and 'score' in ood[domain]:
                noise = np.random.normal(0, get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.AdaptivePriorCalculator._add_ood_noise").get("auto_param_L5058_44", 0.05))  #
5% noise
                ood[domain]['score'] = np.clip(ood[domain]['score'] + noise, get_parameter
_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._add_ood_noise").get("
auto_param_L5059_77", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.Adapt
ivePriorCalculator._add_ood_noise").get("auto_param_L5059_82", 1.0))

        return ood

    def generate_traceability_record(
```

```
        self,
        evidence_dict: dict[str, Any],
        test_type: str,
        result: dict[str, Any],
        seed: int = 42
    ) -> dict[str, Any]:
        """
        PROMPT I-3: Trazabilidad y reproducibilidad

        Con semilla fija, guarda bf_table_version, weights_version,
        snippets textuales con offsets, campos financieros usados.

        METRICS:
        - Re-ejecución con misma semilla produce hash_result idéntico
        - trace_completeness ≥ get_parameter_loader().get("saaaaaa.analysis.derek_beach.Ad
aptivePriorCalculator._add_ood_noise").get("auto_param_L5078_31", 0.95)
        """
        # Fijar semilla para reproducibilidad
        np.random.seed(seed)

        # Construir evidence trace
        evidence_trace = []
        for domain, data in evidence_dict.items():
            if isinstance(data, dict) and 'score' in data:
                trace_item = {
                    'source': domain,
                    'line_span': data.get('line_span', 'unknown'),
                    'transform_before': data.get('raw_value', None),
                    'transform_after': data['score'],
                    'snippet': data.get('snippet', '')[:100]  # Primeros 100 chars
                }
                evidence_trace.append(trace_item)

        # Config hash
        config_str = json.dumps({
            'bf_table_version': self.bf_table.get_version(),
            'calibration_params': self.calibration,
            'domain_weights': self.default_domain_weights,
            'test_type': test_type,
            'seed': seed
        }, sort_keys=True)

        config_hash = hashlib.sha256(config_str.encode()).hexdigest()[:16]

        # Result hash
        result_str = json.dumps(result, sort_keys=True)
        result_hash = hashlib.sha256(result_str.encode()).hexdigest()[:16]

        # Trace completeness
        factors_in_trace = len(evidence_trace)
        total_factors = len([d for d in evidence_dict if isinstance(evidence_dict.get(d),
dict)])
        trace_completeness = factors_in_trace / max(total_factors, 1)

        return {
            'evidence_trace': evidence_trace,
            'hash_config': config_hash,
            'hash_result': result_hash,
            'seed': seed,
            'bf_table_version': self.bf_table.get_version(),
            'weights_version': 'default_vget_parameter_loader().get("saaaaaa.analysis.dere
k_beach.AdaptivePriorCalculator._add_ood_noise").get("auto_param_L5122_41", 1.0)',
            'trace_completeness': trace_completeness,
            'reproducibility_guaranteed': trace_completeness >= get_parameter_loader().get
("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator._add_ood_noise").get("auto_param_L5
124_64", 0.95)
        }
```

```
    @calibrated_method("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.validate_qual
ity_criteria")
    def validate_quality_criteria(self, validation_samples: list[dict[str, Any]]) ->
dict[str, Any]:
        """
        Valida criterios de calidad en conjunto de validación sintética

        QUALITY CRITERIA:
        - BrierScore ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePr
iorCalculator.validate_quality_criteria").get("auto_param_L5133_23", 0.20)
        - ACE ∈ [–get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCa
lculator.validate_quality_criteria").get("auto_param_L5134_18", 0.02), get_parameter_loade
r().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.validate_quality_criteria").
get("auto_param_L5134_24", 0.02)]
        - Cobertura CI95% ∈ [92%, 98%]
        - Monotonicidad verificada
        """
        predictions = []
        actuals = []

        for sample in validation_samples:
            evidence = sample.get('evidence', {})
            actual_label = sample.get('actual_label', get_parameter_loader().get("saaaaaa.
analysis.derek_beach.AdaptivePriorCalculator.validate_quality_criteria").get("auto_param_L
5143_54", 0.5))
            test_type = sample.get('test_type', 'hoop')

            result = self.calculate_likelihood_adaptativo(evidence, test_type)
            predictions.append(result['p_mechanism'])
            actuals.append(actual_label)

        predictions = np.array(predictions)
        actuals = np.array(actuals)

        # 1. Brier Score
        brier_score = np.mean((predictions - actuals) ** 2)
        brier_ok = brier_score <= get_parameter_loader().get("saaaaaa.analysis.derek_beach
.AdaptivePriorCalculator.validate_quality_criteria").get("auto_param_L5155_34", 0.20)

        # 2. ACE (Average Calibration Error)
        # Dividir en bins
        n_bins = 10
        bin_boundaries = np.linspace(0, 1, n_bins + 1)
        ace = get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalcul
ator.validate_quality_criteria").get("ace", 0.0) # Refactored

        for i in range(n_bins):
            bin_mask = (predictions >= bin_boundaries[i]) & (predictions <
bin_boundaries[i + 1])
            if bin_mask.sum() > 0:
                bin_accuracy = actuals[bin_mask].mean()
                bin_confidence = predictions[bin_mask].mean()
                ace += abs(bin_accuracy - bin_confidence) / n_bins

        ace_ok = -get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCa
lculator.validate_quality_criteria").get("auto_param_L5170_18", 0.02) <= ace <= get_parame
ter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.validate_quality_cr
iteria").get("auto_param_L5170_33", 0.02)

        # 3. Cobertura CI95%
        # Simular con bootstrap
        n_bootstrap = 100
        coverage_count = 0

        for _ in range(n_bootstrap):
            idx = np.random.choice(len(predictions), size=len(predictions), replace=True)
            boot_preds = predictions[idx]
            boot_actuals = actuals[idx]
```

```
            # Calcular CI95%
            ci_low = np.percentile(boot_preds, 2.5)
            ci_high = np.percentile(boot_preds, 97.5)

            # Verificar si mean actual está dentro
            actual_mean = boot_actuals.mean()
            if ci_low <= actual_mean <= ci_high:
                coverage_count += 1

        coverage = coverage_count / n_bootstrap
        coverage_ok = get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePri
orCalculator.validate_quality_criteria").get("auto_param_L5192_22", 0.92) <= coverage <= g
et_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.validate_q
uality_criteria").get("auto_param_L5192_42", 0.98)

        # 4. Monotonicidad: verificar que ↑ señales → ¬↓ p_mechanism
        monotonicity_violations = 0

        for i in range(len(validation_samples) - 1):
            current_total = sum(
                validation_samples[i]['evidence'].get(d, {}).get('score', 0)
                for d in ['semantic', 'temporal', 'financial', 'structural']
            )
            next_total = sum(
                validation_samples[i + 1]['evidence'].get(d, {}).get('score', 0)
                for d in ['semantic', 'temporal', 'financial', 'structural']
            )

            if next_total > current_total and predictions[i + 1] < predictions[i]:
                monotonicity_violations += 1

        monotonicity_ok = monotonicity_violations == 0

        return {
            'brier_score': float(brier_score),
            'brier_ok': brier_ok,
            'ace': float(ace),
            'ace_ok': ace_ok,
            'ci95_coverage': float(coverage),
            'coverage_ok': coverage_ok,
            'monotonicity_violations': monotonicity_violations,
            'monotonicity_ok': monotonicity_ok,
            'all_criteria_met': brier_ok and ace_ok and coverage_ok and monotonicity_ok,
            'quality_grade': 'EXCELLENT' if (brier_ok and ace_ok and coverage_ok and
monotonicity_ok) else 'NEEDS_IMPROVEMENT'
        }

# ============================================================================
# AGUJA II: MODELO GENERATIVO JERÁRQUICO
# ============================================================================

class HierarchicalGenerativeModel:
    """
    AGUJA II - Modelo Generativo Jerárquico con inferencia MCMC

    PROMPT II-1: Inferencia jerárquica con incertidumbre
    Estima posterior(mechanism_type, activity_sequence | obs) con MCMC.

    PROMPT II-2: Posterior Predictive Checks + Ablation
    Genera datos simulados desde posterior y compara con observados.

    PROMPT II-3: Independencias y parsimonia
    Verifica d-separaciones y calcula ΔWAIC.

    QUALITY CRITERIA:
    - R-hat ≤ 1.10
    - ESS ≥ 200
```

```
   - entropy/entropy_max < get_parameter_loader().get("saaaaaa.analysis.derek_beach.Adapt
ivePriorCalculator.validate_quality_criteria").get("auto_param_L5245_28", 0.7) para
certeza
   - ppd_p_value ∈ [get_parameter_loader().get("saaaaaa.analysis.derek_beach.AdaptivePrio
rCalculator.validate_quality_criteria").get("auto_param_L5246_21", 0.1), get_parameter_loa
der().get("saaaaaa.analysis.derek_beach.AdaptivePriorCalculator.validate_quality_criteria"
).get("auto_param_L5246_26", 0.9)]
   - ΔWAIC ≤ −2 para preferir jerárquico
   """

   def __init__(self, mechanism_priors: dict[str, float] | None = None) -> None:
      self.logger = logging.getLogger(self.__class__.__name__)

      # Priors débiles para mechanism_type si no se proveen
      self.mechanism_priors = mechanism_priors or {
         'administrativo': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hie
rarchicalGenerativeModel.__init__").get("auto_param_L5255_30", 0.30),
         'tecnico': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierarchic
alGenerativeModel.__init__").get("auto_param_L5256_23", 0.25),
         'financiero': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierarc
hicalGenerativeModel.__init__").get("auto_param_L5257_26", 0.20),
         'politico': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierarchi
calGenerativeModel.__init__").get("auto_param_L5258_24", 0.15),
         'mixto': get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierarchical
GenerativeModel.__init__").get("auto_param_L5259_21", 0.10)
      }

      # Validar que suman ~get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hier
archicalGenerativeModel.__init__").get("auto_param_L5262_29", 1.0)
      prior_sum = sum(self.mechanism_priors.values())
      if abs(prior_sum - get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierar
chicalGenerativeModel.__init__").get("auto_param_L5264_27", 1.0)) > get_parameter_loader()
.get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_
L5264_34", 0.01):
         self.logger.warning(f"Mechanism priors sum to {prior_sum:.3f},
normalizing...")
         self.mechanism_priors = {
            k: v / prior_sum for k, v in self.mechanism_priors.items()
         }

   def infer_mechanism_posterior(
      self,
      observations: dict[str, Any],
      n_iter: int = 500,
      burn_in: int = 100,
      n_chains: int = 2
   ) -> dict[str, Any]:
      """
      PROMPT II-1: Inferencia jerárquica con MCMC

      Estima posterior(mechanism_type, activity_sequence | obs) usando MCMC.

      Args:
         observations: Dict con {verbos, co_ocurrencias, coherence, structural_signals}
         n_iter: Iteraciones MCMC (≥500)
         burn_in: Burn-in iterations (≥100)
         n_chains: Número de cadenas para R-hat (≥2)

      Returns:
         Dict con type_posterior, sequence_mode, coherence_score, entropy, CI95, R-hat,
 ESS
      """
      self.logger.info(f"Starting MCMC inference: {n_iter} iter, {burn_in} burn-in,
{n_chains} chains")

      # Validar observaciones mínimas
      if not observations or 'coherence' not in observations:
         self.logger.warning("Missing observations, using weak priors")
```

```
        observations = observations or {}
        observations.setdefault('coherence', get_parameter_loader().get("saaaaaa.analy
sis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5297_49", 0.5))

    # Ejecutar múltiples cadenas para diagnóstico
    chains = []
    for chain_idx in range(n_chains):
        chain_samples = self._run_mcmc_chain(
            observations, n_iter, burn_in, seed=42 + chain_idx
        )
        chains.append(chain_samples)
        self.logger.debug(f"Chain {chain_idx + 1}/{n_chains} completed:
{len(chain_samples)} samples")

    # Agregar samples de todas las cadenas
    all_samples = []
    for chain in chains:
        all_samples.extend(chain)

    # 1. Type posterior (frecuencias de mechanism_type)
    type_counts = dict.fromkeys(self.mechanism_priors.keys(), 0)
    for sample in all_samples:
        mtype = sample.get('mechanism_type', 'mixto')
        if mtype in type_counts:
            type_counts[mtype] += 1

    total_samples = len(all_samples)
    type_posterior = {
        mtype: count / max(total_samples, 1)
        for mtype, count in type_counts.items()
    }

    # 2. Sequence mode (secuencia más frecuente)
    sequence_mode = self._get_mode_sequence(all_samples)

    # 3. Coherence score (estadísticas)
    coherence_scores = [s.get('coherence', get_parameter_loader().get("saaaaaa.analysi
s.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5330_47", 0.5)) for
s in all_samples]
    coherence_mean = float(np.mean(coherence_scores))
    coherence_std = float(np.std(coherence_scores))

    # 4. Entropy del posterior
    posterior_probs = list(type_posterior.values())
    entropy_posterior = -sum(p * np.log(p + 1e-10) for p in posterior_probs if p > 0)
    max_entropy = np.log(len(self.mechanism_priors))
    normalized_entropy = entropy_posterior / max_entropy if max_entropy > 0 else get_p
arameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__")
.get("auto_param_L5338_85", 0.0)

    # 5. CI95 para coherence
    ci95_low = float(np.percentile(coherence_scores, 2.5))
    ci95_high = float(np.percentile(coherence_scores, 97.5))

    # 6. R-hat aproximado (between-chain variance / within-chain variance)
    r_hat = self._calculate_r_hat(chains)

    # 7. ESS (Effective Sample Size)
    ess = self._calculate_ess(all_samples)

    # 8. Verificar criterios de calidad
    is_uncertain = normalized_entropy > get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5351_44", 0.7)
    criteria_met = {
        'r_hat_ok': r_hat <= 1.10,
        'ess_ok': ess >= 200,
        'entropy_ok': not is_uncertain
    }
```

```
        # Warning si alta incertidumbre
        warning = None
        if is_uncertain:
            warning = f"HIGH_UNCERTAINTY: entropy/entropy_max = {normalized_entropy:.3f} >
 get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel.__in
it__").get("auto_param_L5361_91", 0.7)"
            self.logger.warning(warning)

        return {
            'type_posterior': type_posterior,
            'sequence_mode': sequence_mode,
            'coherence_score': coherence_mean,
            'coherence_std': coherence_std,
            'entropy_posterior': float(entropy_posterior),
            'normalized_entropy': float(normalized_entropy),
            'CI95': (ci95_low, ci95_high),
            'CI95_width': ci95_high - ci95_low,
            'R_hat': float(r_hat),
            'ESS': float(ess),
            'n_samples': total_samples,
            'is_uncertain': is_uncertain,
            'criteria_met': criteria_met,
            'warning': warning
        }

    def _run_mcmc_chain(
        self,
        observations: dict[str, Any],
        n_iter: int,
        burn_in: int,
        seed: int
    ) -> list[dict[str, Any]]:
        """Ejecuta una cadena MCMC con Metropolis-Hastings"""
        np.random.seed(seed)
        samples = []

        # Estado inicial: sample desde prior
        current_type = np.random.choice(
            list(self.mechanism_priors.keys()),
            p=list(self.mechanism_priors.values())
        )
        current_coherence = observations.get('coherence', get_parameter_loader().get("saaa
aaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5397_58",
 0.5))

        for i in range(n_iter):
            # Proponer nuevo mechanism_type
            proposed_type = np.random.choice(list(self.mechanism_priors.keys()))

            # Calcular likelihood ratio
            current_likelihood = self._calculate_likelihood(current_type, observations)
            proposed_likelihood = self._calculate_likelihood(proposed_type, observations)

            # Prior ratio
            prior_ratio = self.mechanism_priors[proposed_type] /
max(self.mechanism_priors[current_type], 1e-10)

            # Acceptance probability (Metropolis-Hastings)
            likelihood_ratio = proposed_likelihood / max(current_likelihood, 1e-10)
            acceptance_prob = min(get_parameter_loader().get("saaaaaa.analysis.derek_beach
.HierarchicalGenerativeModel.__init__").get("auto_param_L5412_34", 1.0), likelihood_ratio
* prior_ratio)

            # Accept/reject
            if np.random.random() < acceptance_prob:
                current_type = proposed_type
```

```
            # Simular coherence con ruido
            simulated_coherence = current_coherence + np.random.normal(0, get_parameter_lo
ader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_
param_L5419_74", 0.05))
            simulated_coherence = np.clip(simulated_coherence, get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5420
_63", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativ
eModel.__init__").get("auto_param_L5420_68", 1.0))

            # Almacenar sample (después de burn-in)
            if i >= burn_in:
                sample = {
                    'mechanism_type': current_type,
                    'coherence': float(simulated_coherence),
                    'iteration': i - burn_in,
                    'chain_seed': seed
                }
                samples.append(sample)

        return samples

    def _calculate_likelihood(
        self,
        mechanism_type: str,
        observations: dict[str, Any]
    ) -> float:
        """Calcula likelihood de observations dado mechanism_type"""
        # Likelihood basado en coherence y structural signals
        coherence = observations.get('coherence', get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5441_50", 0.5))
        structural_signals = observations.get('structural_signals', {})

        # Base likelihood desde prior
        prior = self.mechanism_priors.get(mechanism_type, get_parameter_loader().get("saaa
aaa.analysis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5445_58",
 0.1))

        # Ajuste por coherence (mayor coherence → mayor likelihood)
        coherence_factor = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierar
chicalGenerativeModel.__init__").get("auto_param_L5448_27", 1.0) + coherence

        # Ajuste por señales estructurales específicas del tipo
        structural_match = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hierar
chicalGenerativeModel.__init__").get("structural_match", 0.0) # Refactored
        if mechanism_type == 'administrativo' and structural_signals.get('admin_keywords',
 0) > 0:
            structural_match = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hi
erarchicalGenerativeModel.__init__").get("structural_match", 0.2) # Refactored
        elif mechanism_type == 'financiero' and structural_signals.get('budget_data', 0) >
 0:
            structural_match = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hi
erarchicalGenerativeModel.__init__").get("structural_match", 0.3) # Refactored
        elif mechanism_type == 'tecnico' and structural_signals.get('technical_terms', 0)
> 0:
            structural_match = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hi
erarchicalGenerativeModel.__init__").get("structural_match", 0.25) # Refactored

        likelihood = prior * coherence_factor * (get_parameter_loader().get("saaaaaa.analy
sis.derek_beach.HierarchicalGenerativeModel.__init__").get("auto_param_L5459_49", 1.0) +
structural_match)
        return likelihood

    @calibrated_method("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._get_mode
_sequence")
    def _get_mode_sequence(self, samples: list[dict[str, Any]]) -> str:
        """Obtiene secuencia modal (tipo más frecuente)"""
        type_counts = {}
        for s in samples:
```

```
            mtype = s.get('mechanism_type', 'mixto')
            type_counts[mtype] = type_counts.get(mtype, 0) + 1

        if type_counts:
            return max(type_counts.items(), key=lambda x: x[1])[0]
        return 'mixto'

    @calibrated_method("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculat
e_r_hat")
    def _calculate_r_hat(self, chains: list[list[dict[str, Any]]]) -> float:
        """Calcula Gelman-Rubin R-hat para diagnóstico de convergencia"""
        if len(chains) < 2:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGe
nerativeModel._calculate_r_hat").get("auto_param_L5478_19", 1.0)

        # Extraer coherence de cada cadena
        chain_means = []
        chain_vars = []

        for chain in chains:
            coherences = [s.get('coherence', get_parameter_loader().get("saaaaaa.analysis.
derek_beach.HierarchicalGenerativeModel._calculate_r_hat").get("auto_param_L5485_45",
0.5)) for s in chain]
            if len(coherences) > 0:
                chain_means.append(np.mean(coherences))
                chain_vars.append(np.var(coherences, ddof=1))

        if len(chain_means) < 2:
            return get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGe
nerativeModel._calculate_r_hat").get("auto_param_L5491_19", 1.0)

        # Between-chain variance (B)
        n = len(chains[0])  # samples per chain
        B = np.var(chain_means, ddof=1) * n

        # Within-chain variance (W)
        W = np.mean(chain_vars)

        # R-hat estimator
        if W > 0:
            var_plus = ((n - 1) / n) * W + (1 / n) * B
            r_hat = np.sqrt(var_plus / W)
        else:
            r_hat = get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalG
enerativeModel._calculate_r_hat").get("r_hat", 1.0) # Refactored

        return float(r_hat)

    @calibrated_method("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculat
e_ess")
    def _calculate_ess(self, samples: list[dict[str, Any]]) -> float:
        """Calcula Effective Sample Size (simplificado)"""
        n = len(samples)

        # Estimar autocorrelación
        coherences = np.array([s.get('coherence', get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_param_L5515_50",
0.5)) for s in samples])

        if len(coherences) < 2:
            return n

        # Lag-1 autocorrelation
        mean_coh = np.mean(coherences)
        var_coh = np.var(coherences)

        if var_coh > 0:
            lag1_autocorr = np.mean(
```

```
                (coherences[:-1] - mean_coh) * (coherences[1:] - mean_coh)
            ) / var_coh
        else:
            lag1_autocorr = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hiera
rchicalGenerativeModel._calculate_ess").get("lag1_autocorr", 0.0) # Refactored

        # ESS approximation
        ess = n / (1 + 2 * max(0, lag1_autocorr))
        return float(ess)

    def posterior_predictive_check(
        self,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, Any]:
        """
        PROMPT II-2: Posterior Predictive Checks + Ablation

        Genera datos simulados desde posterior y compara con observados.
        Realiza ablation de pasos de secuencia.

        Args:
            posterior_samples: Samples del posterior MCMC
            observed_data: Datos observados reales

        Returns:
            Dict con ppd_p_value, distance_metric, ablation_curve, criteria_met
        """
        self.logger.info("Running posterior predictive checks...")

        # 1. Generar datos predictivos desde posterior
        n_ppd_samples = min(100, len(posterior_samples))
        ppd_samples = []

        for _i in range(n_ppd_samples):
            sample_idx = np.random.randint(0, len(posterior_samples))
            posterior_sample = posterior_samples[sample_idx]

            # Simular coherence desde distribución posterior
            simulated_coherence = posterior_sample.get('coherence', get_parameter_loader()
.get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_
param_L5564_68", 0.5)) + np.random.normal(0, get_parameter_loader().get("saaaaaa.analysis.
derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_param_L5564_95", 0.05))
            simulated_coherence = np.clip(simulated_coherence, get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_param
_L5565_63", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGen
erativeModel._calculate_ess").get("auto_param_L5565_68", 1.0))
            ppd_samples.append(simulated_coherence)

        ppd_samples = np.array(ppd_samples)

        # 2. Comparar con observado usando KS test
        observed_coherence = observed_data.get('coherence', get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_param_L5
571_60", 0.5))

        # KS test: comparar distribución PPD con punto observado
        from scipy.stats import kstest
        ks_stat, ppd_p_value = kstest(ppd_samples, lambda x: 0 if x < observed_coherence
else 1)
        ppd_p_value = float(ppd_p_value)

        # 3. Ablation de secuencia
        ablation_curve = self._ablation_analysis(posterior_samples, observed_data)

        # 4. Verificar criterios
        ppd_ok = get_parameter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGene
rativeModel._calculate_ess").get("auto_param_L5582_17", 0.1) <= ppd_p_value <= get_paramet
```

```
er_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess")
.get("auto_param_L5582_39", 0.9)
        ablation_ok = all(delta >= -get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.HierarchicalGenerativeModel._calculate_ess").get("auto_param_L5583_36", 0.05) for delta
 in ablation_curve.values())  # Tolerancia -5%

        criteria_met = {
            'ppd_p_value_ok': ppd_ok,
            'ablation_ok': ablation_ok
        }

        # Recomendación
        if ppd_ok and ablation_ok:
            recommendation = 'accept'
        else:
            recommendation = 'rebaja_posterior'
            self.logger.warning(f"PPC failed: ppd_p={ppd_p_value:.3f},
ablation_ok={ablation_ok}")

        return {
            'ppd_p_value': ppd_p_value,
            'ppd_samples_mean': float(np.mean(ppd_samples)),
            'ppd_samples_std': float(np.std(ppd_samples)),
            'distance_metric': 'KS',
            'ks_statistic': float(ks_stat),
            'ablation_curve': ablation_curve,
            'criteria_met': criteria_met,
            'recommendation': recommendation
        }

    def _ablation_analysis(
        self,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, float]:
        """Mide caída en coherence al quitar pasos de secuencia"""
        baseline_coherence = np.mean([s.get('coherence', get_parameter_loader().get("saaaa
aa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess").get("auto_param_L5614
_57", 0.5)) for s in posterior_samples])

        # Simular ablación de pasos clave
        # En práctica real, esto requeriría re-ejecutar modelo sin ciertos steps
        ablation_deltas = {
            'remove_step_diagnostic': baseline_coherence - (baseline_coherence * get_param
eter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess
").get("auto_param_L5619_81", 0.95)),  # -5%
            'remove_step_planning': baseline_coherence - (baseline_coherence * get_paramet
er_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess")
.get("auto_param_L5620_79", 0.85)),    # -15%
            'remove_step_execution': baseline_coherence - (baseline_coherence * get_parame
ter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess"
).get("auto_param_L5621_80", 0.90)),   # -10%
            'remove_step_monitoring': baseline_coherence - (baseline_coherence * get_param
eter_loader().get("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculate_ess
").get("auto_param_L5622_81", 0.97))   # -3%
        }

        return ablation_deltas

    def verify_conditional_independence(
        self,
        dag: nx.DiGraph,
        independence_tests: list[tuple[str, str, list[str]]] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT II-3: Independencias y parsimonia

        Verifica d-separaciones implicadas por el DAG.
```

```
    Calcula ΔWAIC entre modelo jerárquico vs. nulo.

    Args:
        dag: NetworkX DiGraph del modelo causal
        independence_tests: Lista de tuplas (X, Y, Z) para test X ⊥ Y | Z

    Returns:
        Dict con independence_tests, delta_waic, model_preference, criteria_met
    """
    self.logger.info("Verifying conditional independencies...")

    # 1. Tests de independencia (d-separación)
    test_results = []

    if independence_tests is None:
        # Generar tests automáticamente si no se proveen
        independence_tests = self._generate_independence_tests(dag)

    for x, y, z_set in independence_tests:
        try:
            # Verificar d-separación en DAG
            is_independent = nx.d_separated(dag, {x}, {y}, set(z_set))
            test_results.append({
                'test': f"{x} ⊥ {y} | {{{', '.join(z_set)}}}",
                'x': x,
                'y': y,
                'z': z_set,
                'passed': is_independent
            })
        except Exception as e:
            self.logger.warning(f"Independence test failed: {x} ⊥ {y} | {z_set} -
{e}")
            test_results.append({
                'test': f"{x} ⊥ {y} | {{{', '.join(z_set)}}}",
                'x': x,
                'y': y,
                'z': z_set,
                'passed': False,
                'error': str(e)
            })

    tests_passed = sum(1 for t in test_results if t['passed'])

    # 2. Calcular ΔWAIC (simplificado)
    # En práctica real: usar librería como arviz para WAIC calculation
    delta_waic = self._calculate_waic_difference(dag)

    # 3. Verificar criterios
    independence_ok = tests_passed >= 2
    waic_ok = delta_waic <= -2.0

    # 4. Preferencia de modelo
    if independence_ok and waic_ok:
        model_preference = 'hierarchical'
    elif not waic_ok:
        model_preference = 'inconclusive'
    else:
        model_preference = 'null'

    criteria_met = {
        'independence_ok': independence_ok,
        'waic_ok': waic_ok
    }

    return {
        'independence_tests': test_results,
        'tests_passed': tests_passed,
        'tests_total': len(test_results),
```

```
            'delta_waic': float(delta_waic),
            'model_preference': model_preference,
            'criteria_met': criteria_met
        }

    def _generate_independence_tests(
        self,
        dag: nx.DiGraph,
        n_tests: int = 3
    ) -> list[tuple[str, str, list[str]]]:
        """Genera tests de independencia automáticamente desde DAG"""
        tests = []
        nodes = list(dag.nodes())

        if len(nodes) < 3:
            return tests

        # Generar tests de forma heurística
        for _ in range(min(n_tests, len(nodes) - 2)):
            # Seleccionar nodos aleatorios
            x, y = np.random.choice(nodes, size=2, replace=False)

            # Z: padres comunes o mediadores
            z_candidates = set(dag.predecessors(x)) | set(dag.predecessors(y))
            z_set = list(z_candidates)[:2]  # Máximo 2 nodos en conditioning set

            if x != y:
                tests.append((x, y, z_set))

        return tests

    @calibrated_method("saaaaaa.analysis.derek_beach.HierarchicalGenerativeModel._calculat
e_waic_difference")
    def _calculate_waic_difference(self, dag: nx.DiGraph) -> float:
        """
        Calcula ΔWAIC = WAIC_hierarchical - WAIC_null (simplificado)

        En producción: usar arviz.waic() con trace real de PyMC/Stan
        """
        # Heurística: modelos jerárquicos con más estructura (edges) son preferidos
        n_edges = dag.number_of_edges()
        dag.number_of_nodes()

        # Penalización por complejidad
        complexity_penalty = n_edges * get_parameter_loader().get("saaaaaa.analysis.derek_
beach.HierarchicalGenerativeModel._calculate_waic_difference").get("auto_param_L5746_39",
0.5)

        # WAIC aproximado
        waic_hierarchical = -5get_parameter_loader().get("saaaaaa.analysis.derek_beach.Hie
rarchicalGenerativeModel._calculate_waic_difference").get("auto_param_L5749_30", 0.0) -
n_edges * 2  # Mejor fit con más estructura
        waic_null = -45.0  # Modelo nulo sin estructura

        delta_waic = waic_hierarchical - waic_null + complexity_penalty

        return delta_waic

# ============================================================================
# AGUJA III: AUDITOR CONTRAFACTUAL BAYESIANO
# ============================================================================

class BayesianCounterfactualAuditor:
    """
    AGUJA III - Auditor Contrafactual con SCM y do-calculus

    PROMPT III-1: Construcción de SCM y queries gemelas
    Construye SCM={DAG, f_i} y responde omission_impact, sufficiency_test, necessity_test.
```

```
    PROMPT III-2: Riesgo sistémico y priorización
    Agrega riesgos, propaga incertidumbre, calcula priority.

    PROMPT III-3: Refutación, negativos y cordura do(.)
    Ejecuta controles negativos, pruebas placebo, sanity checks.

    QUALITY CRITERIA:
    - Consistencia de signos factual/contrafactual
    - effect_stability: Δeffect ≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach
.HierarchicalGenerativeModel._calculate_waic_difference").get("auto_param_L5775_34", 0.15)
 al variar priors ±10%
    - negative_controls: mediana |efecto| ≤ get_parameter_loader().get("saaaaaa.analysis.d
erek_beach.HierarchicalGenerativeModel._calculate_waic_difference").get("auto_param_L5776_
44", 0.05)
    - sanity_violations: 0
    """

    def __init__(self) -> None:
        self.logger = logging.getLogger(self.__class__.__name__)
        self.scm: dict[str, Any] | None = None

    def construct_scm(
        self,
        dag: nx.DiGraph,
        structural_equations: dict[str, callable] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-1: Construcción de SCM

        Construye SCM = {DAG, f_i} desde grafo y ecuaciones estructurales.

        Args:
            dag: NetworkX DiGraph (debe ser acíclico)
            structural_equations: Dict {node: function} para f_i

        Returns:
            SCM con DAG validado y funciones estructurales

        Raises:
            ValueError: Si DAG no es acíclico
        """
        self.logger.info(f"Constructing SCM with {dag.number_of_nodes()} nodes,
{dag.number_of_edges()} edges")

        # 1. Validar que DAG es acíclico
        if not nx.is_directed_acyclic_graph(dag):
            raise ValueError("DAG must be acyclic for SCM construction. Use cycle
detection first.")

        # 2. Crear ecuaciones por defecto si no se proveen
        if structural_equations is None:
            structural_equations = self._create_default_equations(dag)
            self.logger.info(f"Created {len(structural_equations)} default structural
equations")

        # 3. Construir SCM
        scm = {
            'dag': dag,
            'equations': structural_equations,
            'nodes': list(dag.nodes()),
            'edges': list(dag.edges()),
            'topological_order': list(nx.topological_sort(dag))
        }

        self.scm = scm
        self.logger.info("✓ SCM constructed successfully")
        return scm
```

```
    @calibrated_method("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create
_default_equations")
    def _create_default_equations(self, dag: nx.DiGraph) -> dict[str, callable]:
        """Crea ecuaciones estructurales lineales por defecto"""
        equations = {}

        for node in dag.nodes():
            parents = list(dag.predecessors(node))

            if not parents:
                # Nodo raíz: variable exógena U
                def root_eq(noise=get_parameter_loader().get("saaaaaa.analysis.derek_beach
.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L5838_34",
0.0), node_name=node):
                    return get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anCounterfactualAuditor._create_default_equations").get("auto_param_L5839_27", 0.5) +
noise  # Prior neutral + ruido
                equations[node] = root_eq
            else:
                # Nodo con padres: función lineal
                def child_eq(parent_values, noise=get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param
_L5843_50", 0.0), node_name=node, n_parents=len(parents)):
                    if isinstance(parent_values, dict):
                        return sum(parent_values.values()) / max(n_parents, 1) + noise
                    return get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesi
anCounterfactualAuditor._create_default_equations").get("auto_param_L5846_27", 0.5) +
noise
                equations[node] = child_eq

        return equations

    def counterfactual_query(
        self,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-1: Queries gemelas (omission, sufficiency, necessity)

        Evalúa:
        - Factual: P(Y | evidence)
        - Counterfactual: P(Y | do(X=x), evidence)
        - Causal effect, sufficiency, necessity

        Args:
            intervention: {nodo: valor} para do(.) operation
            target: Nodo objetivo Y
            evidence: Evidencia observada (opcional)

        Returns:
            Dict con p_factual, p_counterfactual, causal_effect, is_sufficient,
is_necessary
        """
        if self.scm is None:
            raise ValueError("SCM must be constructed first. Call construct_scm().")

        evidence = evidence or {}

        self.logger.debug(f"Counterfactual query: intervention={intervention},
target={target}")

        # 1. Factual: P(Y | evidence)
        p_factual = self._evaluate_factual(target, evidence)

        # 2. Counterfactual: P(Y | do(X=x), evidence)
```

```
        p_counterfactual = self._evaluate_counterfactual(target, intervention, evidence)

        # 3. Causal effect
        causal_effect = p_counterfactual - p_factual

        # 4. Sufficiency test: ¿do(X=1) → Y=1?
        intervention_node = list(intervention.keys())[0] if intervention else None
        if intervention_node:
            p_y_given_do_x1 = self._evaluate_counterfactual(target, {intervention_node: ge
t_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._crea
te_default_equations").get("auto_param_L5892_88", 1.0)}, {})
            is_sufficient = p_y_given_do_x1 > get_parameter_loader().get("saaaaaa.analysis
.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L58
93_46", 0.7)
        else:
            is_sufficient = False

        # 5. Necessity test: ¿do(X=0) → Y=0?
        if intervention_node:
            p_y_given_do_x0 = self._evaluate_counterfactual(target, {intervention_node: ge
t_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._crea
te_default_equations").get("auto_param_L5899_88", 0.0)}, {})
            is_necessary = p_y_given_do_x0 < get_parameter_loader().get("saaaaaa.analysis.
derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L590
0_45", 0.3)
        else:
            is_necessary = False

        # 6. Consistencia de signos
        signs_consistent = (
            (causal_effect >= 0 and p_counterfactual >= p_factual) or
            (causal_effect < 0 and p_counterfactual < p_factual)
        )

        # 7. Effect stability
        stability = self._test_effect_stability(intervention, target, evidence)

        return {
            'p_factual': float(np.clip(p_factual, get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param
_L5914_50", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounter
factualAuditor._create_default_equations").get("auto_param_L5914_55", 1.0))),
            'p_counterfactual': float(np.clip(p_counterfactual, get_parameter_loader().get
("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").g
et("auto_param_L5915_64", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.B
ayesianCounterfactualAuditor._create_default_equations").get("auto_param_L5915_69",
1.0))),
            'causal_effect': float(causal_effect),
            'is_sufficient': is_sufficient,
            'is_necessary': is_necessary,
            'signs_consistent': signs_consistent,
            'effect_stability': float(stability),
            'effect_stable': stability <= get_parameter_loader().get("saaaaaa.analysis.der
ek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L5921_4
2", 0.15)
        }

    def _evaluate_factual(
        self,
        target: str,
        evidence: dict[str, float]
    ) -> float:
        """Evalúa P(target | evidence) propagando hacia adelante en DAG"""
        if target in evidence:
            return evidence[target]

        dag = self.scm['dag']
        equations = self.scm['equations']
```

```
        topological_order = self.scm['topological_order']

        # Evaluar nodos en orden topológico
        computed_values = evidence.copy()

        for node in topological_order:
            if node in computed_values:
                continue

            parents = list(dag.predecessors(node))

            if not parents:
                # Nodo raíz
                computed_values[node] = equations[node](noise=get_parameter_loader().get("
saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get
("auto_param_L5948_62", 0.0))
            else:
                # Evaluar padres primero
                parent_values = {}
                for parent in parents:
                    if parent not in computed_values:
                        computed_values[parent] = self._evaluate_factual(parent, evidence)
                    parent_values[parent] = computed_values[parent]

                # Aplicar ecuación estructural
                try:
                    computed_values[node] = equations[node](parent_values, noise=get_param
eter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_defa
ult_equations").get("auto_param_L5959_81", 0.0))
                except:
                    # Fallback
                    computed_values[node] = sum(parent_values.values()) /
max(len(parent_values), 1)

        return float(np.clip(computed_values.get(target, get_parameter_loader().get("saaaa
aa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("aut
o_param_L5964_57", 0.5)), get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesia
nCounterfactualAuditor._create_default_equations").get("auto_param_L5964_63", 0.0), get_pa
rameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_d
efault_equations").get("auto_param_L5964_68", 1.0)))

    def _evaluate_counterfactual(
        self,
        target: str,
        intervention: dict[str, float],
        evidence: dict[str, float]
    ) -> float:
        """Evalúa P(target | do(intervention), evidence) con DAG mutilado"""
        # Crear DAG mutilado: quitar aristas hacia nodos intervenidos
        dag_mutilated = self.scm['dag'].copy()

        for node in intervention:
            in_edges = list(dag_mutilated.in_edges(node))
            dag_mutilated.remove_edges_from(in_edges)

        # Guardar SCM original
        original_scm = self.scm.copy()

        # Crear SCM mutilado temporalmente
        self.scm = {
            'dag': dag_mutilated,
            'equations': self.scm['equations'],
            'nodes': self.scm['nodes'],
            'edges': list(dag_mutilated.edges()),
            'topological_order': list(nx.topological_sort(dag_mutilated))
        }

        # Combinar evidence con intervention (intervention tiene prioridad)
```

```
        combined_evidence = {**evidence, **intervention}

        # Evaluar en SCM mutilado
        result = self._evaluate_factual(target, combined_evidence)

        # Restaurar SCM original
        self.scm = original_scm

        return result

    def _test_effect_stability(
        self,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None,
        n_perturbations: int = 5
    ) -> float:
        """Testa estabilidad al variar priors/ecuaciones ±10%"""
        evidence = evidence or {}

        # Efecto baseline
        baseline_result = self.counterfactual_query(intervention, target, evidence)
        baseline_effect = baseline_result['causal_effect']

        # Perturbar y medir variación
        perturbed_effects = []

        for _ in range(n_perturbations):
            perturbation_factor = np.random.uniform(get_parameter_loader().get("saaaaaa.an
alysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_par
am_L6021_52", 0.9), 1.1)  # ±10%

            # Perturbar valores de evidencia
            perturbed_evidence = {
                k: v * perturbation_factor for k, v in evidence.items()
            }

            # Re-evaluar
            try:
                result = self.counterfactual_query(intervention, target,
perturbed_evidence)
                perturbed_effects.append(result['causal_effect'])
            except:
                perturbed_effects.append(baseline_effect)

        # Máxima variación
        max_variation = max(abs(e - baseline_effect) for e in perturbed_effects) if
perturbed_effects else get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCo
unterfactualAuditor._create_default_equations").get("auto_param_L6036_107", 0.0)

        return max_variation

    def aggregate_risk_and_prioritize(
        self,
        omission_score: float,
        insufficiency_score: float,
        unnecessity_score: float,
        causal_effect: float,
        feasibility: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Baye
sianCounterfactualAuditor._create_default_equations").get("auto_param_L6046_29", 0.8),
        cost: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCou
nterfactualAuditor._create_default_equations").get("auto_param_L6047_22", 1.0)
    ) -> dict[str, Any]:
        """
        PROMPT III-2: Riesgo sistémico y priorización con incertidumbre

        Fórmulas:
        - risk = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterf
```

```
actualAuditor._create_default_equations").get("auto_param_L6053_17", 0.50)·omission + get_
parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create
_default_equations").get("auto_param_L6053_33", 0.35)·insufficiency + get_parameter_loader
().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equatio
ns").get("auto_param_L6053_54", 0.15)·unnecessity
        - priority = |effect|·feasibility/(cost+ε)·(1−uncertainty)

        Args:
            omission_score: Riesgo de omisión de mecanismo [0,1]
            insufficiency_score: Insuficiencia del mecanismo [0,1]
            unnecessity_score: Mecanismo innecesario [0,1]
            causal_effect: Efecto causal estimado
            feasibility: Factibilidad de intervención [0,1]
            cost: Costo relativo (>0)

        Returns:
            Dict con risk_score, success_probability, priority, recommendations
        """
        # 1. Componentes de riesgo
        risk_components = {
            'omission': float(np.clip(omission_score, get_parameter_loader().get("saaaaaa.
analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_p
aram_L6069_54", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCou
nterfactualAuditor._create_default_equations").get("auto_param_L6069_59", 1.0))),
            'insufficiency': float(np.clip(insufficiency_score, get_parameter_loader().get
("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").g
et("auto_param_L6070_64", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.B
ayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6070_69",
1.0))),
            'unnecessity': float(np.clip(unnecessity_score, get_parameter_loader().get("sa
aaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("
auto_param_L6071_60", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayes
ianCounterfactualAuditor._create_default_equations").get("auto_param_L6071_65", 1.0)))
        }

        # 2. Riesgo agregado
        risk_score = (
            get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactua
lAuditor._create_default_equations").get("auto_param_L6076_12", 0.50) *
risk_components['omission'] +
            get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactua
lAuditor._create_default_equations").get("auto_param_L6077_12", 0.35) *
risk_components['insufficiency'] +
            get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactua
lAuditor._create_default_equations").get("auto_param_L6078_12", 0.15) *
risk_components['unnecessity']
        )
        risk_score = float(np.clip(risk_score, get_parameter_loader().get("saaaaaa.analysi
s.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6
080_47", 0.0), get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfac
tualAuditor._create_default_equations").get("auto_param_L6080_52", 1.0)))

        # 3. Success probability con incertidumbre
        success_mean = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCo
unterfactualAuditor._create_default_equations").get("auto_param_L6083_23", 1.0) -
risk_score

        # Incertidumbre: mayor riesgo → mayor uncertainty
        success_std = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCou
nterfactualAuditor._create_default_equations").get("auto_param_L6086_22", 0.05) + get_para
meter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_def
ault_equations").get("auto_param_L6086_29", 0.10) * risk_score  # Entre 5% y 15%

        # CI95 para success
        ci95_low = max(get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCo
unterfactualAuditor._create_default_equations").get("auto_param_L6089_23", 0.0),
success_mean - 1.96 * success_std)
        ci95_high = min(get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianC
```

```
ounterfactualAuditor._create_default_equations").get("auto_param_L6090_24", 1.0),
success_mean + 1.96 * success_std)

        success_probability = {
            'mean': float(success_mean),
            'std': float(success_std),
            'CI95': (float(ci95_low), float(ci95_high))
        }

        # 4. Prioridad
        uncertainty = success_std
        epsilon = 1e-6

        priority = (
            abs(causal_effect) *
            feasibility /
            (cost + epsilon) *
            (get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactu
alAuditor._create_default_equations").get("auto_param_L6106_13", 1.0) - uncertainty)
        )
        priority = float(priority)

        # 5. Recomendaciones ordenadas
        recommendations = []

        if risk_score > get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianC
ounterfactualAuditor._create_default_equations").get("auto_param_L6113_24", 0.7):
            recommendations.append("CRITICAL_RISK: Immediate intervention required")
        elif risk_score > get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesia
nCounterfactualAuditor._create_default_equations").get("auto_param_L6115_26", 0.4):
            recommendations.append("MEDIUM_RISK: Close monitoring required")
        else:
            recommendations.append("LOW_RISK: Routine surveillance")

        if risk_components['omission'] > get_parameter_loader().get("saaaaaa.analysis.dere
k_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6120_41
", 0.6):
            recommendations.append("HIGH_OMISSION_RISK: Key mechanism may be missing")

        if risk_components['insufficiency'] > get_parameter_loader().get("saaaaaa.analysis
.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L61
23_46", 0.5):
            recommendations.append("INSUFFICIENCY_DETECTED: Mechanism alone insufficient")

        if priority > get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCou
nterfactualAuditor._create_default_equations").get("auto_param_L6126_22", 0.5):
            recommendations.append("HIGH_PRIORITY: Optimal intervention candidate")
        elif priority < get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianC
ounterfactualAuditor._create_default_equations").get("auto_param_L6128_24", 0.2):
            recommendations.append("LOW_PRIORITY: Consider alternative interventions")

        # 6. Verificar criterios de calidad
        ci95_valid = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCoun
terfactualAuditor._create_default_equations").get("auto_param_L6132_21", 0.0) <= ci95_low
<= ci95_high <= get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfa
ctualAuditor._create_default_equations").get("auto_param_L6132_53", 1.0)
        priority_monotonic = priority >= 0
        risk_in_range = get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianC
ounterfactualAuditor._create_default_equations").get("auto_param_L6134_24", 0.0) <=
risk_score <= get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfact
ualAuditor._create_default_equations").get("auto_param_L6134_45", 1.0)

        criteria_met = {
            'ci95_valid': ci95_valid,
            'priority_monotonic': priority_monotonic,
            'risk_in_range': risk_in_range
        }
```

```
        return {
            'risk_components': risk_components,
            'risk_score': risk_score,
            'success_probability': success_probability,
            'priority': priority,
            'recommendations': sorted(recommendations, reverse=True),
            'criteria_met': criteria_met
        }

    def refutation_and_sanity_checks(
        self,
        dag: nx.DiGraph,
        target: str,
        treatment: str,
        confounders: list[str] | None = None
    ) -> dict[str, Any]:
        """
        PROMPT III-3: Refutación, negativos y cordura do(.)

        Ejecuta:
        1. Controles negativos: nodos irrelevantes → |efecto| ≤ get_parameter_loader().get
("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").g
et("auto_param_L6162_64", 0.05)
        2. Pruebas placebo: permuta edges no causales
        3. Sanity checks: añadir cofactores no reduce P(Y|do(X=1))

        Args:
            dag: Grafo causal
            target: Nodo objetivo Y
            treatment: Nodo de tratamiento X
            confounders: Lista de cofactores

        Returns:
            Dict con negative_controls, placebo_effect, sanity_violations, recommendation
        """
        confounders = confounders or []

        self.logger.info("Running refutation and sanity checks...")

        # 1. CONTROLES NEGATIVOS: nodos irrelevantes
        irrelevant_nodes = [
            n for n in dag.nodes()
            if n not in (target, treatment) and not nx.has_path(dag, n, target)
        ]

        negative_effects = []
        for node in irrelevant_nodes[:5]:  # Máximo 5 controles
            try:
                intervention = {node: get_parameter_loader().get("saaaaaa.analysis.derek_b
each.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6188_38",
1.0)}
                result = self.counterfactual_query(intervention, target, {})
                effect = abs(result['causal_effect'])
                negative_effects.append(effect)
            except Exception as e:
                self.logger.warning(f"Negative control failed for {node}: {e}")

        median_negative_effect = float(np.median(negative_effects)) if negative_effects
else get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAudito
r._create_default_equations").get("auto_param_L6195_93", 0.0)
        negative_controls_ok = median_negative_effect <= get_parameter_loader().get("saaaa
aa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("aut
o_param_L6196_57", 0.05)

        # 2. PRUEBA PLACEBO: permuta edges no causales
        placebo_dag = dag.copy()
        non_causal_edges = [
            (u, v) for u, v in dag.edges()
```

```
            if u != treatment and v != target
        ]

        placebo_effect = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Bayesian
CounterfactualAuditor._create_default_equations").get("placebo_effect", 0.0) # Refactored
        if non_causal_edges:
            # Permutar una arista
            edge_to_remove = non_causal_edges[0]
            placebo_dag.remove_edge(*edge_to_remove)

            # Medir efecto en DAG permutado
            scm_backup = self.scm
            try:
                self.construct_scm(placebo_dag)
                result = self.counterfactual_query({treatment: get_parameter_loader().get(
"saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").ge
t("auto_param_L6215_63", 1.0)}, target, {})
                placebo_effect = abs(result['causal_effect'])
            except Exception as e:
                self.logger.warning(f"Placebo test failed: {e}")
            finally:
                self.scm = scm_backup

        placebo_ok = placebo_effect <= get_parameter_loader().get("saaaaaa.analysis.derek_
beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6222_39",
 0.05)

        # 3. SANITY CHECKS: añadir cofactores activos no debe reducir P(Y|do(X=1))
        sanity_violations = []

        # Baseline: do(X=1)
        try:
            baseline_result = self.counterfactual_query({treatment: get_parameter_loader()
.get("saaaaaa.analysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations
").get("auto_param_L6229_68", 1.0)}, target, {})
            baseline_p = baseline_result['p_counterfactual']

            # Con cofactores
            for confounder in confounders[:2]:  # Máximo 2
                if confounder in dag.nodes():
                    result_with_conf = self.counterfactual_query(
                        {treatment: get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6236_36",
1.0)},
                        target,
                        {confounder: get_parameter_loader().get("saaaaaa.analysis.derek_be
ach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6238_37",
1.0)}
                    )
                    p_with_conf = result_with_conf['p_counterfactual']

                    # Verificar que no reduce significativamente
                    if p_with_conf < baseline_p - get_parameter_loader().get("saaaaaa.anal
ysis.derek_beach.BayesianCounterfactualAuditor._create_default_equations").get("auto_param
_L6243_50", 0.10):
                        sanity_violations.append({
                            'confounder': confounder,
                            'baseline_p': float(baseline_p),
                            'p_with_confounder': float(p_with_conf),
                            'violation': f"Adding {confounder} reduced P(Y|do(X)) by
{baseline_p - p_with_conf:.3f}"
                        })
        except Exception as e:
            self.logger.error(f"Sanity checks failed: {e}")

        sanity_ok = len(sanity_violations) == 0

        # 4. DECISIÓN FINAL
```

```
        all_checks_passed = negative_controls_ok and placebo_ok and sanity_ok

        if not all_checks_passed:
            recommendation = "DEGRADE_ALL: Require DAG revision - observación prioritaria"
            self.logger.error(recommendation)
        else:
            recommendation = "ACCEPT: All refutation tests passed"
            self.logger.info(recommendation)

        return {
            'negative_controls': {
                'effects': [float(e) for e in negative_effects],
                'median': median_negative_effect,
                'passed': negative_controls_ok,
                'criterion': '≤ get_parameter_loader().get("saaaaaa.analysis.derek_beach.B
ayesianCounterfactualAuditor._create_default_equations").get("auto_param_L6270_32", 0.05)'
            },
            'placebo_effect': {
                'effect': float(placebo_effect),
                'passed': placebo_ok,
                'criterion': '≈ 0'
            },
            'sanity_violations': sanity_violations,
            'sanity_passed': sanity_ok,
            'all_checks_passed': all_checks_passed,
            'recommendation': recommendation
        }

def main() -> int:
    """CLI entry point"""
    parser = argparse.ArgumentParser(
        description="CDAF v2.0 - Framework de Deconstrucción y Auditoría Causal",
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Ejemplo de uso:
  python cdaf_framework.py documento.pdf --output-dir resultados/ --policy-code P1

Configuración:
  El framework busca config.yaml en el directorio actual.
  Use --config-file para especificar una ruta alternativa.
        """
    )

    parser.add_argument(
        "pdf_path",
        type=Path,
        help="Ruta al archivo PDF del Plan de Desarrollo Territorial"
    )

    parser.add_argument(
        "--output-dir",
        type=Path,
        default=Path("resultados_analisis"),
        help="Directorio de salida para los artefactos (default: resultados_analisis/)"
    )

    parser.add_argument(
        "--policy-code",
        type=str,
        required=True,
        help="Código de política para nombrar los artefactos (ej: P1, PDT_2024)"
    )

    parser.add_argument(
        "--config-file",
        type=Path,
        default=Path(DEFAULT_CONFIG_FILE),
        help=f"Ruta al archivo de configuración YAML (default: {DEFAULT_CONFIG_FILE})"
```

```
    )

    parser.add_argument(
        "--log-level",
        choices=["DEBUG", "INFO", "WARNING", "ERROR"],
        default="INFO",
        help="Nivel de logging (default: INFO)"
    )

    parser.add_argument(
        "--pdet",
        action="store_true",
        help="Indica si el municipio es PDET (activa validación especial)"
    )

    args = parser.parse_args()

    # Validate inputs
    if not args.pdf_path.exists():
        print(f"ERROR: Archivo PDF no encontrado: {args.pdf_path}")
        return 1

    # Initialize framework
    try:
        framework = CDAFFramework(args.config_file, args.output_dir, args.log_level)

        # Configure PDET if specified
        if args.pdet and framework.dnp_validator:
            framework.dnp_validator.es_municipio_pdet = True
            framework.logger.info("Modo PDET activado - Validación especial habilitada")
    except Exception as e:
        print(f"ERROR: No se pudo inicializar el framework: {e}")
        return 1

    # Process document
    success = framework.process_document(args.pdf_path, args.policy_code)

    return 0 if success else 1

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

class DerekBeachProducer:
    """
    Producer wrapper for Derek Beach causal analysis with registry exposure

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

    Version: get_parameter_loader().get("saaaaaa.analysis.derek_beach.BayesianCounterfactu
alAuditor._create_default_equations").get("auto_param_L6373_13", 1.0).0
    Producer Type: Causal Mechanism Analysis
    """

    def __init__(self) -> None:
        """Initialize producer"""
        self.logger = logging.getLogger(self.__class__.__name__)
        self.logger.info("DerekBeachProducer initialized")

    # ========================================================================
    # EVIDENTIAL TESTS API
    # ========================================================================


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.classify_test_type")
    def classify_test_type(self, necessity: float, sufficiency: float) -> TestType:
        """Classify evidential test type based on necessity and sufficiency"""
```

```
        return BeachEvidentialTest.classify_test(necessity, sufficiency)

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

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_hoop_test")
    def is_hoop_test(self, test_type: TestType) -> bool:
        """Check if test is hoop test"""
        return test_type == "hoop_test"

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_smoking_gun")
    def is_smoking_gun(self, test_type: TestType) -> bool:
        """Check if test is smoking gun"""
        return test_type == "smoking_gun"


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_doubly_decisive")
    def is_doubly_decisive(self, test_type: TestType) -> bool:
        """Check if test is doubly decisive"""
        return test_type == "doubly_decisive"

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_straw_in_wind")
    def is_straw_in_wind(self, test_type: TestType) -> bool:
        """Check if test is straw in wind"""
        return test_type == "straw_in_wind"

    # ========================================================================
    # HIERARCHICAL GENERATIVE MODEL API
    # ========================================================================

    def create_hierarchical_model(
        self,
        mechanism_priors: dict[str, float] | None = None
    ) -> HierarchicalGenerativeModel:
        """Create hierarchical generative model"""
        return HierarchicalGenerativeModel(mechanism_priors)

    def infer_mechanism_posterior(
        self,
        model: HierarchicalGenerativeModel,
        observations: dict[str, Any],
        n_iter: int = 500,
        burn_in: int = 100,
        n_chains: int = 2
    ) -> dict[str, Any]:
        """Infer mechanism posterior using MCMC"""
        return model.infer_mechanism_posterior(
            observations, n_iter, burn_in, n_chains
        )


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_type_posterior")
    def get_type_posterior(self, inference: dict[str, Any]) -> dict[str, float]:
        """Extract type posterior from inference"""
        return inference.get("type_posterior", {})


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_sequence_mode")
    def get_sequence_mode(self, inference: dict[str, Any]) -> str:
```

```
        """Extract sequence mode from inference"""
        return inference.get("sequence_mode", "")


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_coherence_score")
    def get_coherence_score(self, inference: dict[str, Any]) -> float:
        """Extract coherence score from inference"""
        return inference.get("coherence_score", get_parameter_loader().get("saaaaaa.analys
is.derek_beach.DerekBeachProducer.get_coherence_score").get("auto_param_L6460_48", 0.0))

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_r_hat")
    def get_r_hat(self, inference: dict[str, Any]) -> float:
        """Extract R-hat convergence diagnostic"""
        return inference.get("R_hat", get_parameter_loader().get("saaaaaa.analysis.derek_b
each.DerekBeachProducer.get_r_hat").get("auto_param_L6465_38", 1.0))

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_ess")
    def get_ess(self, inference: dict[str, Any]) -> float:
        """Extract effective sample size"""
        return inference.get("ESS", get_parameter_loader().get("saaaaaa.analysis.derek_bea
ch.DerekBeachProducer.get_ess").get("auto_param_L6470_36", 0.0))

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_inference_uncer
tain")
    def is_inference_uncertain(self, inference: dict[str, Any]) -> bool:
        """Check if inference has high uncertainty"""
        return inference.get("is_uncertain", False)

    # ========================================================================
    # POSTERIOR PREDICTIVE CHECKS API
    # ========================================================================

    def posterior_predictive_check(
        self,
        model: HierarchicalGenerativeModel,
        posterior_samples: list[dict[str, Any]],
        observed_data: dict[str, Any]
    ) -> dict[str, Any]:
        """Run posterior predictive checks"""
        return model.posterior_predictive_check(posterior_samples, observed_data)

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_ppd_p_value")
    def get_ppd_p_value(self, ppc: dict[str, Any]) -> float:
        """Extract posterior predictive p-value"""
        return ppc.get("ppd_p_value", get_parameter_loader().get("saaaaaa.analysis.derek_b
each.DerekBeachProducer.get_ppd_p_value").get("auto_param_L6493_38", 0.0))


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_ablation_curve")
    def get_ablation_curve(self, ppc: dict[str, Any]) -> dict[str, float]:
        """Extract ablation curve from PPC"""
        return ppc.get("ablation_curve", {})

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_ppc_recommenda
tion")
    def get_ppc_recommendation(self, ppc: dict[str, Any]) -> str:
        """Extract recommendation from PPC"""
        return ppc.get("recommendation", "")

    # ========================================================================
    # CONDITIONAL INDEPENDENCE API
    # ========================================================================

    def verify_conditional_independence(
        self,
        model: HierarchicalGenerativeModel,
        dag: nx.DiGraph,
        independence_tests: list[tuple[str, str, list[str]]] | None = None
```

```
    ) -> dict[str, Any]:
        """Verify conditional independencies in DAG"""
        return model.verify_conditional_independence(dag, independence_tests)

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_independence_t
ests")
    def get_independence_tests(self, verification: dict[str, Any]) -> list[dict[str,
Any]]:
        """Extract independence tests from verification"""
        return verification.get("independence_tests", [])

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_delta_waic")
    def get_delta_waic(self, verification: dict[str, Any]) -> float:
        """Extract delta WAIC from verification"""
        return verification.get("delta_waic", get_parameter_loader().get("saaaaaa.analysis
.derek_beach.DerekBeachProducer.get_delta_waic").get("auto_param_L6526_46", 0.0))


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_model_preference")
    def get_model_preference(self, verification: dict[str, Any]) -> str:
        """Extract model preference from verification"""
        return verification.get("model_preference", "inconclusive")

    # ============================================================================
    # COUNTERFACTUAL AUDITOR API
    # ============================================================================

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.create_auditor")
    def create_auditor(self) -> BayesianCounterfactualAuditor:
        """Create Bayesian counterfactual auditor"""
        return BayesianCounterfactualAuditor()

    def construct_scm(
        self,
        auditor: BayesianCounterfactualAuditor,
        dag: nx.DiGraph,
        structural_equations: dict[str, callable] | None = None
    ) -> dict[str, Any]:
        """Construct structural causal model"""
        return auditor.construct_scm(dag, structural_equations)

    def counterfactual_query(
        self,
        auditor: BayesianCounterfactualAuditor,
        intervention: dict[str, float],
        target: str,
        evidence: dict[str, float] | None = None
    ) -> dict[str, Any]:
        """Execute counterfactual query"""
        return auditor.counterfactual_query(intervention, target, evidence)


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_causal_effect")
    def get_causal_effect(self, query: dict[str, Any]) -> float:
        """Extract causal effect from query"""
        return query.get("causal_effect", get_parameter_loader().get("saaaaaa.analysis.der
ek_beach.DerekBeachProducer.get_causal_effect").get("auto_param_L6564_42", 0.0))

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_sufficient")
    def is_sufficient(self, query: dict[str, Any]) -> bool:
        """Check if mechanism is sufficient"""
        return query.get("is_sufficient", False)

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_necessary")
    def is_necessary(self, query: dict[str, Any]) -> bool:
        """Check if mechanism is necessary"""
        return query.get("is_necessary", False)
```

```
    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.is_effect_stable")
    def is_effect_stable(self, query: dict[str, Any]) -> bool:
        """Check if effect is stable"""
        return query.get("effect_stable", False)

    # ========================================================================
    # RISK AGGREGATION API
    # ========================================================================

    def aggregate_risk(
        self,
        auditor: BayesianCounterfactualAuditor,
        omission_score: float,
        insufficiency_score: float,
        unnecessity_score: float,
        causal_effect: float,
        feasibility: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.Dere
kBeachProducer.is_effect_stable").get("auto_param_L6592_29", 0.8),
        cost: float = get_parameter_loader().get("saaaaaa.analysis.derek_beach.DerekBeachP
roducer.is_effect_stable").get("auto_param_L6593_22", 1.0)
    ) -> dict[str, Any]:
        """Aggregate risk and calculate priority"""
        return auditor.aggregate_risk_and_prioritize(
            omission_score,
            insufficiency_score,
            unnecessity_score,
            causal_effect,
            feasibility,
            cost
        )

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_risk_score")
    def get_risk_score(self, aggregation: dict[str, Any]) -> float:
        """Extract risk score from aggregation"""
        return aggregation.get("risk_score", get_parameter_loader().get("saaaaaa.analysis.
derek_beach.DerekBeachProducer.get_risk_score").get("auto_param_L6608_45", 0.0))

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_success_probab
ility")
    def get_success_probability(self, aggregation: dict[str, Any]) -> dict[str, float]:
        """Extract success probability from aggregation"""
        return aggregation.get("success_probability", {})

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_priority")
    def get_priority(self, aggregation: dict[str, Any]) -> float:
        """Extract priority from aggregation"""
        return aggregation.get("priority", get_parameter_loader().get("saaaaaa.analysis.de
rek_beach.DerekBeachProducer.get_priority").get("auto_param_L6618_43", 0.0))


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_recommendations")
    def get_recommendations(self, aggregation: dict[str, Any]) -> list[str]:
        """Extract recommendations from aggregation"""
        return aggregation.get("recommendations", [])

    # ========================================================================
    # REFUTATION API
    # ========================================================================

    def refutation_checks(
        self,
        auditor: BayesianCounterfactualAuditor,
        dag: nx.DiGraph,
        target: str,
        treatment: str,
        confounders: list[str] | None = None
    ) -> dict[str, Any]:
        """Execute refutation and sanity checks"""
```

```
        return auditor.refutation_and_sanity_checks(
            dag, target, treatment, confounders
        )

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_negative_contr
ols")
    def get_negative_controls(self, refutation: dict[str, Any]) -> dict[str, Any]:
        """Extract negative controls from refutation"""
        return refutation.get("negative_controls", {})


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_placebo_effect")
    def get_placebo_effect(self, refutation: dict[str, Any]) -> dict[str, Any]:
        """Extract placebo effect from refutation"""
        return refutation.get("placebo_effect", {})

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_sanity_violati
ons")
    def get_sanity_violations(self, refutation: dict[str, Any]) -> list[dict[str, Any]]:
        """Extract sanity violations from refutation"""
        return refutation.get("sanity_violations", [])


@calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.all_checks_passed")
    def all_checks_passed(self, refutation: dict[str, Any]) -> bool:
        """Check if all refutation checks passed"""
        return refutation.get("all_checks_passed", False)

    @calibrated_method("saaaaaa.analysis.derek_beach.DerekBeachProducer.get_refutation_rec
ommendation")
    def get_refutation_recommendation(self, refutation: dict[str, Any]) -> str:
        """Extract recommendation from refutation"""
        return refutation.get("recommendation", "")

===== FILE: src/saaaaaa/analysis/enhance_recommendation_rules.py =====
#!/usr/bin/env python3
"""
Script to enhance recommendation rules with 7 advanced features:
1. Template parameterization
2. Rule execution logic
3. Measurable indicators
4. Unambiguous time horizons
5. Testable verification
6. Cost tracking
7. Authority mapping
"""

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


def enhance_template(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 1: Eliminate Hardcoded Template Strings
    Replace with template_id and template_params
    """
    template = rule.get('template', {})
    rule_id = rule.get('rule_id', '')
    level = rule.get('level', '')

    # Extract parameters from template strings
    template_params = {}

    if level == 'MICRO':
        when = rule.get('when', {})
```

```
        template_params = {
            'pa_id': when.get('pa_id', 'PA01'),
            'dim_id': when.get('dim_id', 'DIM01'),
            'question_id': 'Q001'  # Would be derived from context
        }
    elif level == 'MESO':
        when = rule.get('when', {})
        template_params = {
            'cluster_id': when.get('cluster_id', 'CL01')
        }

    # Create enhanced template with ID
    enhanced_template = copy.deepcopy(template)
    enhanced_template['template_id'] = f"TPL-{rule_id}"
    enhanced_template['template_params'] = template_params

    return enhanced_template

def add_execution_logic(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 2: Add Rule Execution Logic
    """
    when = rule.get('when', {})
    level = rule.get('level', '')

    # Build trigger condition string
    trigger_parts = []
    if level == 'MICRO':
        pa_id = when.get('pa_id', '')
        dim_id = when.get('dim_id', '')
        score_lt = when.get('score_lt', 1.65)
        trigger_parts.append(f"score < {score_lt}")
        trigger_parts.append(f"pa_id = '{pa_id}'")
        trigger_parts.append(f"dim_id = '{dim_id}'")
    elif level == 'MESO':
        cluster_id = when.get('cluster_id', '')
        score_band = when.get('score_band', '')
        if score_band:
            trigger_parts.append(f"score_band = '{score_band}'")
        if cluster_id:
            trigger_parts.append(f"cluster_id = '{cluster_id}'")
    elif level == 'MACRO':
        macro_band = when.get('macro_band', '')
        if macro_band:
            trigger_parts.append(f"macro_band = '{macro_band}'")

    trigger_condition = " AND ".join(trigger_parts) if trigger_parts else "true"

    return {
        "trigger_condition": trigger_condition,
        "blocking": False,
        "auto_apply": False,
        "requires_approval": True,
        "approval_roles": ["Secretaría de Planeación", "Secretaría de Hacienda"]
    }

def enhance_indicator(indicator: dict[str, Any], rule_id: str, level: str) -> dict[str,
Any]:
    """
    Feature 3: Make Indicators Measurable
    """
    enhanced = copy.deepcopy(indicator)

    # Add formula based on indicator type
    indicator.get('name', '')
    if 'proporción' in indicator.get('unit', ''):
        enhanced['formula'] = 'COUNT(compliant_items) / COUNT(total_items)'
        enhanced['acceptable_range'] = [0.6, 1.0]
```

```
    elif 'porcentaje' in indicator.get('unit', ''):
        enhanced['formula'] = '(achieved / target) * 100'
        enhanced['acceptable_range'] = [60.0, 100.0]
    else:
        enhanced['formula'] = 'SUM(verified_artifacts)'
        enhanced['acceptable_range'] = [indicator.get('target', 1) * 0.7,
indicator.get('target', 1)]

    # Add measurement metadata
    enhanced['baseline_measurement_date'] = '2024-01-01'
    enhanced['measurement_frequency'] = 'mensual'
    enhanced['data_source'] = 'Sistema de Seguimiento de Planes (SSP)'
    enhanced['data_source_query'] = f"SELECT COUNT(*) FROM indicators WHERE indicator_id =
 '{rule_id}-IND'"
    enhanced['responsible_measurement'] = 'Oficina de Planeación Municipal'
    enhanced['escalation_if_below'] = enhanced['acceptable_range'][0]

    return enhanced

def enhance_horizon(horizon: dict[str, str], rule_id: str) -> dict[str, Any]:
    """
    Feature 4: Define Unambiguous Time Horizons
    """
    # Map T0, T1, T2, T3 to actual durations
    duration_map = {
        'T0': 0,
        'T1': 6,   # 6 months
        'T2': 12,  # 12 months
        'T3': 24   # 24 months
    }

    start = horizon.get('start', 'T0')
    end = horizon.get('end', 'T1')

    duration_months = duration_map.get(end, 6) - duration_map.get(start, 0)

    # Create milestones
    milestones = []
    if duration_months >= 6:
        milestones.append({
            "name": "Inicio de implementación",
            "offset_months": 1,
            "deliverables": ["Plan de trabajo aprobado"],
            "verification_required": True
        })
    if duration_months >= 12:
        milestones.append({
            "name": "Revisión intermedia",
            "offset_months": duration_months // 2,
            "deliverables": ["Informe de avance"],
            "verification_required": True
        })
    milestones.append({
        "name": "Entrega final",
        "offset_months": duration_months,
        "deliverables": ["Todos los productos esperados"],
        "verification_required": True
    })

    return {
        "start": start,
        "end": end,
        "start_type": "plan_approval_date",
        "duration_months": duration_months,
        "milestones": milestones,
        "dependencies": [],
        "critical_path": duration_months <= 6
    }
```

```
def enhance_verification(verification: list[str], rule_id: str) -> list[dict[str, Any]]:
    """
    Feature 5: Make Verification Testable
    """
    enhanced_verifications = []

    for idx, artifact_text in enumerate(verification, 1):
        # Determine artifact type
        artifact_type = "DOCUMENT"
        if any(word in artifact_text.lower() for word in ['sistema', 'repositorio',
'registro', 'base de datos']):
            artifact_type = "SYSTEM_STATE"

        # Create structured verification
        ver_obj = {
            "id": f"VER-{rule_id}-{idx:03d}",
            "type": artifact_type,
            "artifact": artifact_text,
            "format": "PDF" if artifact_type == "DOCUMENT" else "DATABASE_QUERY",
            "required_sections": ["Objetivo", "Alcance", "Resultados"] if artifact_type ==
 "DOCUMENT" else [],
            "approval_required": True,
            "approver": "Secretaría de Planeación",
            "due_date": "T1",
            "automated_check": artifact_type == "SYSTEM_STATE"
        }

        # Add validation for system states
        if artifact_type == "SYSTEM_STATE":
            ver_obj["validation_query"] = f"SELECT COUNT(*) FROM artifacts WHERE
artifact_id = '{ver_obj['id']}'"
            ver_obj["pass_condition"] = "COUNT(*) >= 1"

        enhanced_verifications.append(ver_obj)

    return enhanced_verifications

def add_budget(rule: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 6: Integrate Cost Tracking
    """
    level = rule.get('level', '')

    # Estimate costs based on level and complexity
    if level == 'MICRO':
        base_cost = 45_000_000  # COP
    elif level == 'MESO':
        base_cost = 150_000_000
    elif level == 'MACRO':
        base_cost = 500_000_000
    else:
        base_cost = 50_000_000

    # Cost breakdown
    personal_cost = int(base_cost * 0.55)
    consultancy_cost = int(base_cost * 0.30)
    technology_cost = int(base_cost * 0.15)

    return {
        "estimated_cost_cop": base_cost,
        "cost_breakdown": {
            "personal": personal_cost,
            "consultancy": consultancy_cost,
            "technology": technology_cost
        },
        "funding_sources": [
            {
```

```
                "source": "SGP - Sistema General de Participaciones",
                "amount": int(base_cost * 0.60),
                "confirmed": False
            },
            {
                "source": "Recursos Propios",
                "amount": int(base_cost * 0.40),
                "confirmed": False
            }
        ],
        "fiscal_year": 2025
    }

def enhance_responsible(responsible: dict[str, Any]) -> dict[str, Any]:
    """
    Feature 7: Map Authority for Accountability
    """
    enhanced = copy.deepcopy(responsible)

    # Add legal mandate
    entity = responsible.get('entity', '')
    if 'Mujer' in entity:
        legal_mandate = "Ley 1257 de 2008 - Normas para la prevención de violencias contra
 la mujer"
    elif 'Planeación' in entity:
        legal_mandate = "Ley 152 de 1994 - Ley Orgánica del Plan de Desarrollo"
    elif 'Hacienda' in entity:
        legal_mandate = "Ley 819 de 2003 - Responsabilidad Fiscal"
    else:
        legal_mandate = "Estatuto Orgánico Municipal"

    enhanced['legal_mandate'] = legal_mandate

    # Add approval chain
    enhanced['approval_chain'] = [
        {
            "level": 1,
            "role": "Director/Coordinador de Programa",
            "decision": "Aprueba plan de trabajo"
        },
        {
            "level": 2,
            "role": "Secretario/a de la entidad responsable",
            "decision": "Aprueba presupuesto y recursos"
        },
        {
            "level": 3,
            "role": "Secretaría de Planeación",
            "decision": "Valida coherencia con PDM"
        },
        {
            "level": 4,
            "role": "Alcalde Municipal",
            "decision": "Aprobación final (si aplica)"
        }
    ]

    # Add escalation path
    enhanced['escalation_path'] = {
        "threshold_days_delay": 15,
        "escalate_to": "Secretaría de Planeación",
        "final_escalation": "Despacho del Alcalde",
        "consequences": ["Revisión presupuestal", "Reasignación de responsables"]
    }

    return enhanced

def enhance_rule(rule: dict[str, Any]) -> dict[str, Any]:
```

```
    """Enhance a single rule with all 7 features"""
    enhanced_rule = copy.deepcopy(rule)

    # 1. Template parameterization
    enhanced_rule['template'] = enhance_template(rule)

    # 2. Execution logic
    enhanced_rule['execution'] = add_execution_logic(rule)

    # 3. Measurable indicators
    if 'indicator' in enhanced_rule['template']:
        enhanced_rule['template']['indicator'] = enhance_indicator(
            enhanced_rule['template']['indicator'],
            rule.get('rule_id', ''),
            rule.get('level', '')
        )

    # 4. Unambiguous time horizons
    if 'horizon' in enhanced_rule['template']:
        enhanced_rule['template']['horizon'] = enhance_horizon(
            enhanced_rule['template']['horizon'],
            rule.get('rule_id', '')
        )

    # 5. Testable verification
    if 'verification' in enhanced_rule['template']:
        enhanced_rule['template']['verification'] = enhance_verification(
            enhanced_rule['template']['verification'],
            rule.get('rule_id', '')
        )

    # 6. Budget tracking
    enhanced_rule['budget'] = add_budget(rule)

    # 7. Authority mapping
    if 'responsible' in enhanced_rule['template']:
        enhanced_rule['template']['responsible'] = enhance_responsible(
            enhanced_rule['template']['responsible']
        )

    return enhanced_rule

def main() -> None:
    """Main enhancement process"""
    # Delegate to factory for I/O operations
    from .factory import load_json, save_json

    # Load existing rules
    rules_path = Path('config/recommendation_rules.json')
    rules_data = load_json(rules_path)

    print(f"Loaded {len(rules_data['rules'])} rules from {rules_path}")

    # Enhance all rules
    enhanced_rules = []
    for i, rule in enumerate(rules_data['rules'], 1):
        try:
            enhanced = enhance_rule(rule)
            enhanced_rules.append(enhanced)
            if i % 10 == 0:
                print(f"Enhanced {i}/{len(rules_data['rules'])} rules...")
        except Exception as e:
            print(f"Error enhancing rule {rule.get('rule_id', 'UNKNOWN')}: {e}")
            enhanced_rules.append(rule)  # Keep original if enhancement fails

    # Create enhanced data structure
    enhanced_data = {
        'version': '2.0',  # Increment version
```

```
        'enhanced_features': [
            'template_parameterization',
            'execution_logic',
            'measurable_indicators',
            'unambiguous_time_horizons',
            'testable_verification',
            'cost_tracking',
            'authority_mapping'
        ],
        'rules': enhanced_rules
    }

    # Save enhanced rules
    output_path = Path('config/recommendation_rules_enhanced.json')
    save_json(enhanced_data, output_path)

    print(f"\nEnhanced {len(enhanced_rules)} rules saved to {output_path}")
    print(f"Original file preserved at {rules_path}")

    # Show sample enhanced rule
    if enhanced_rules:
        print("\n=== Sample Enhanced Rule ===")
        print(json.dumps(enhanced_rules[0], indent=2, ensure_ascii=False)[:2000])
        print("...")

# Note: Main entry point removed to maintain I/O boundary separation.
# For usage examples, see examples/ directory.

===== FILE: src/saaaaaa/analysis/factory.py =====
"""
Factory Layer for Analysis Module I/O Operations

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

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

import csv
import json
import logging
from functools import lru_cache
from pathlib import Path
from typing import Any
from saaaaaa.core.calibration.decorators import calibrated_method

try:
    import yaml
except ImportError:
    yaml = None

try:
    import fitz  # PyMuPDF
except ImportError:
    fitz = None

try:
    import pdfplumber
except ImportError:
    pdfplumber = None

try:
    import spacy
except ImportError:
    spacy = None

logger = logging.getLogger(__name__)
```

```
_PROJECT_ROOT = Path(__file__).resolve().parents[3]
_CALIBRATION_SEARCH_PATHS: tuple[Path, ...] = (
    _PROJECT_ROOT / "config" / "calibraciones",
    _PROJECT_ROOT / "config",
    _PROJECT_ROOT,
)

# ============================================================================
# JSON I/O OPERATIONS
# ============================================================================

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

    Args:
        file_path: Path to JSON file

    Returns:
        Dict containing the loaded JSON data

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

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

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

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

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

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

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

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

# ============================================================================
# YAML I/O OPERATIONS
# ============================================================================

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

    Args:
        file_path: Path to YAML file

    Returns:
        Dict containing the loaded YAML data
```

Raises:
    ImportError: If PyYAML is not installed