```
class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
    """
    Identifies implementation risks and mitigation measures.

    Methods (from D2-Q4):
    - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
    - PDETMunicipalPlanAnalyzer.sensitivity_analysis
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Bayesian risk inference
        risk_inference = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
        )

        # Step 2: Sensitivity analysis
        sensitivity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
            risks=risk_inference
        )

        # Step 3: Risk interpretation
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            inference=risk_inference
        )

        # Step 4: Compute robustness metrics
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            sensitivity=sensitivity
        )
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            robustness=robustness
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            sensitivity=sensitivity
        )

        # Step 5: Audit systemic risks
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 6: Aggregate and prioritize risks
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            risks=risk_inference
        )

        # Step 7: Refutation and sanity checks
        refutation_checks = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            aggregation=risk_aggregation
        )
```

```
        # Step 8: Additional sensitivity analysis
        adaptive_sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=risk_inference
        )

        raw_evidence = {
            "operational_risks": [r for r in risk_inference if r.get("type") ==
"operational"],
            "social_risks": [r for r in risk_inference if r.get("type") == "social"],
            "security_risks": [r for r in risk_inference if r.get("type") == "security"],
            "mitigation_measures": (risk_interpretation or {}).get("mitigations", []),
            "risk_priorities": risk_aggregation,
            "robustness_metrics": {
                "robustness_value": robustness,
                "e_value": e_value
            },
            "sensitivity_analysis": sensitivity,
            "systemic_risks": systemic_risk_audit,
            "validation_checks": refutation_checks,
            "sensitivity_interpretation": sensitivity_interpretation,
            "adaptive_sensitivity": adaptive_sensitivity,
            "risk_interpretation": risk_interpretation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "risks_identified": len(risk_inference or []),
                "mitigations_proposed": len((risk_interpretation or {}).get("mitigations",
 []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
    """
    Evaluates strategic coherence: complementarity and logical sequence.

    Methods (from D2-Q5):
    - PolicyContradictionDetector._detect_logical_incompatibilities
    - PolicyContradictionDetector._calculate_coherence_metrics
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._calculate_graph_fragmentation
    - OperationalizationAuditor.audit_sequence_logic
    - BayesianMechanismInference._calculate_coherence_factor
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Detect logical incompatibilities
        incompatibilities = self._execute_method(
            "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
        )

        # Step 2: Calculate coherence metrics
        coherence_metrics = self._execute_method(
            "PolicyContradictionDetector", "_calculate_coherence_metrics", context
```

```
        )
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )
        graph_fragmentation = self._execute_method(
            "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
        )

        # Step 3: Audit sequence logic
        sequence_audit = self._execute_method(
            "OperationalizationAuditor", "audit_sequence_logic", context
        )

        # Step 4: Calculate coherence factors
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            metrics=coherence_metrics
        )
        causal_coherence_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 5: Adaptive likelihood calculation
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
            coherence=causal_coherence_score
        )

        raw_evidence = {
            "complementarity_evidence": coherence_metrics.get("complementarity", []),
            "sequential_logic": sequence_audit,
            "logical_incompatibilities": incompatibilities,
            "coherence_scores": {
                "overall_coherence": coherence_metrics,
                "objective_alignment": objective_alignment,
                "causal_coherence": causal_coherence_score,
                "coherence_factor": coherence_factor
            },
            "graph_metrics": {
                "fragmentation": graph_fragmentation
            },
            "adaptive_likelihood": adaptive_likelihood
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "incompatibilities_found": len(incompatibilities),
                "coherence_score": coherence_metrics.get("score", 0)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 3: PRODUCTS & OUTPUTS
# ============================================================================

class D3_Q1_IndicatorQualityValidator(BaseExecutor):
    """
    Validates indicator quality: baseline, target, source of verification.

    Methods (from D3-Q1):
```

```
    - PDETMunicipalPlanAnalyzer._score_indicators
    - OperationalizationAuditor.audit_evidence_traceability
    - CausalInferenceSetup.assign_probative_value
    - BeachEvidentialTest.apply_test_logic
    - TextMiningEngine.diagnose_critical_links
    - IndustrialPolicyProcessor._extract_metadata
    - IndustrialPolicyProcessor._calculate_quality_score
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score indicators
        indicator_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )

        # Step 2: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            indicators=indicator_scores
        )

        # Step 3: Assign probative value
        probative_values = self._execute_method(
            "CausalInferenceSetup", "assign_probative_value", context,
            indicators=indicator_scores
        )

        # Step 4: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context,
            indicators=indicator_scores
        )

        # Step 5: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 6: Extract and score metadata
        metadata = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context
        )
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context,
            metadata=metadata
        )

        # Step 7: Generate traceability record
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            indicators=indicator_scores
        )

        raw_evidence = {
            "indicators_with_baseline": [i for i in indicator_scores if
i.get("has_baseline")],
            "indicators_with_target": [i for i in indicator_scores if
i.get("has_target")],
            "indicators_with_source": [i for i in indicator_scores if
i.get("has_source")],
            "indicator_quality_scores": indicator_scores,
            "traceability": traceability_audit,
            "probative_values": probative_values,
            "evidential_strength": evidential_tests,
            "critical_links": critical_links,
```

```
            "overall_quality_score": quality_score,
            "traceability_record": traceability_record
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_indicators": len(indicator_scores or []),
                "complete_indicators": len([i for i in indicator_scores
                    if i.get("has_baseline") and i.get("has_target") and
i.get("has_source")]),
                "critical_links_assessed": len(critical_links or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
    """
    DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY — Analyzes proportionality of targets to the
diagnosed universe using canonical D3 notation.
    Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes
 tests, and semantic indicator quality.

    Methods (from D3-Q2):
    - AdvancedDAGValidator._calculate_bayesian_posterior
    - AdvancedDAGValidator._calculate_confidence_interval
    - AdaptivePriorCalculator._adjust_domain_weights
    - PDETMunicipalPlanAnalyzer._get_spanish_stopwords
    - BayesianMechanismInference._log_refactored_components
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - FinancialAuditor._calculate_sufficiency
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - PDETMunicipalPlanAnalyzer._assess_financial_sustainability
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    - IndustrialPolicyProcessor._calculate_quality_score
    - TeoriaCambio._generar_sugerencias_internas
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._indicator_to_dict
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - IndustrialPolicyProcessor._compile_pattern_registry
    - IndustrialPolicyProcessor._build_point_patterns
    - IndustrialPolicyProcessor._empty_result
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Financial feasibility snapshot and indicator quality
        financial_feasibility = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        spanish_stopwords = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords", context
        )
        funding_sources = self._execute_method(
```

```
    "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
    financial_indicators=financial_feasibility.get("financial_indicators", []),
    tables=context.get("tables", [])
)
financial_component = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
    financial_analysis=financial_feasibility
)
pattern_registry = self._execute_method(
    "IndustrialPolicyProcessor", "_compile_pattern_registry", context
)
point_patterns = self._execute_method(
    "IndustrialPolicyProcessor", "_build_point_patterns", context
)
empty_policy_result = self._execute_method(
    "IndustrialPolicyProcessor", "_empty_result", context
)
dedup_tables = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
    tables=context.get("tables", [])
)
# Type-safe indicator extraction: explicit None, not wrong-typed {}
first_indicator = None
if isinstance(financial_feasibility.get("financial_indicators", []), list):
    inds = financial_feasibility.get("financial_indicators", [])
    if inds and isinstance(inds[0], dict):
        first_indicator = inds[0]

# Pass None explicitly when no indicator exists, maintaining type contract
indicator_dict = None
if first_indicator is not None:
    indicator_dict = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
        ind=first_indicator
    )
proportionality_recommendations = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
    analysis_results={
        "financial_analysis": financial_feasibility,
        "quality_score": quality_score
    }
)

# Step 1: Calculate sufficiency
sufficiency_calc = self._execute_method(
    "FinancialAuditor", "_calculate_sufficiency", context
)

# Step 2: Test sufficiency and necessity of targets
sufficiency_test = self._execute_method(
    "BayesianMechanismInference", "_test_sufficiency", context
)
necessity_test = self._execute_method(
    "BayesianMechanismInference", "_test_necessity", context
)

# Step 3: Assess financial sustainability
sustainability_assessment = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability", context
)
risk_interpretation = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
    risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)
)

# Step 4: Calculate adaptive likelihood
adaptive_likelihood = self._execute_method(
    "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context
```

```
        )
        domain_scores = {
            "structural": sufficiency_calc.get("coverage_ratio", 0.0),
            "financial": financial_feasibility.get("sustainability_score", 0.0),
            "semantic": indicator_quality if isinstance(indicator_quality, (int, float))
else 0.0
        }
        adjusted_weights = self._execute_method(
            "AdaptivePriorCalculator", "_adjust_domain_weights", context,
            domain_scores=domain_scores
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D3": {"dimension_confidence":
domain_scores.get("structural", 0.0)}}
        )

        # Step 6: Generate internal suggestions
        internal_suggestions = self._execute_method(
            "TeoriaCambio", "_generar_sugerencias_internas", context
        )
        # Bayesian posterior diagnostics for proportionality evidence
        posterior_probability = self._execute_method(
            "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,
            likelihood=sufficiency_calc.get("coverage_ratio", 0.5),
            prior=0.5
        )
        confidence_interval = self._execute_method(
            "AdvancedDAGValidator", "_calculate_confidence_interval", context,
            s=int(sufficiency_calc.get("covered_targets", 0)),
            n=max(1, int(sufficiency_calc.get("targets_total",
len(context.get("product_targets", [])))))),
            conf=0.95
        )
        self._execute_method(
            "BayesianMechanismInference", "_log_refactored_components", context
        )

        raw_evidence = {
            "target_population_size": context.get("diagnosed_universe", 0),
            "product_targets": context.get("product_targets", []),
            "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
            "dosage_analysis": sufficiency_calc.get("dosage", {}),
            "sufficiency_test": sufficiency_test,
            "necessity_test": necessity_test,
            "sustainability": sustainability_assessment,
            "financial_feasibility": financial_feasibility,
            "indicator_quality": indicator_quality,
            "risk_interpretation": risk_interpretation,
            "proportionality_score": quality_score,
            "recommendations": internal_suggestions,
            "stopwords_spanish": spanish_stopwords,
            "funding_sources_analysis": funding_sources,
            "financial_component_score": financial_component,
            "pattern_registry": pattern_registry,
            "point_patterns": point_patterns,
            "empty_policy_result": empty_policy_result,
            "avg_confidence": avg_confidence,
            "deduplicated_tables": dedup_tables,
            "indicator_sample": indicator_dict,
            "proportionality_recommendations": proportionality_recommendations,
            "adjusted_domain_weights": adjusted_weights,
            "posterior_proportionality": posterior_probability,
            "coverage_interval": confidence_interval
        }

        return {
            "executor_id": self.executor_id,
```

```
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "targets_analyzed": len(context.get("product_targets", [])),
                "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
                "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q3_TraceabilityValidator(BaseExecutor):
    """
    DIM03_Q03_TRACEABILITY_BUDGET_ORG — Validates budgetary and organizational
traceability of products under canonical D3 notation.
    Epistemic mix: structural budget tracing, organizational semantics, and accountability
 synthesis.

    Methods executed (in order):
    Step 1: Budget matching - FinancialAuditor._match_program_to_node
    Step 2: Goal-budget matching - FinancialAuditor._match_goal_to_budget
    Step 3: Responsibility extraction -
PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
    Step 4: Entity consolidation - PDETMunicipalPlanAnalyzer._consolidate_entities
    Step 5: Entity identification -
PDETMunicipalPlanAnalyzer.identify_responsible_entities
    Step 6: Clarity scoring - PDETMunicipalPlanAnalyzer._score_responsibility_clarity
    Step 7: Document processing - PolicyAnalysisEmbedder.process_document
    Step 8: Query generation - PolicyAnalysisEmbedder._generate_query_from_pdq
    Step 9: Semantic search - PolicyAnalysisEmbedder.semantic_search
    Step 10: MMR diversification - PolicyAnalysisEmbedder._apply_mmr
    Step 11: Semantic cube baseline - SemanticAnalyzer._empty_semantic_cube
    Step 12: Policy domain classification - SemanticAnalyzer._classify_policy_domain
    Step 13: Cross-cutting themes - SemanticAnalyzer._classify_cross_cutting_themes
    Step 14: Value chain classification - SemanticAnalyzer._classify_value_chain_link
    Step 15: Segment vectorization - SemanticAnalyzer._vectorize_segments
    Step 16: Segment processing - SemanticAnalyzer._process_segment
    Step 17: Semantic complexity - SemanticAnalyzer._calculate_semantic_complexity
    Step 18: Evidence confidence - IndustrialPolicyProcessor._compute_evidence_confidence
    Step 19: Entity serialization - PDETMunicipalPlanAnalyzer._entity_to_dict (loop)
    Step 20: Traceability record - AdaptivePriorCalculator.generate_traceability_record
    Step 21: PDQ report - PolicyAnalysisEmbedder.generate_pdq_report
    Step 22: Accountability matrix - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Match programs to budget nodes
        program_matches = self._execute_method(
            "FinancialAuditor", "_match_program_to_node", context
        )
        goal_budget_matches = self._execute_method(
            "FinancialAuditor", "_match_goal_to_budget", context,
            programs=program_matches
        )

        # Step 2: Extract responsibility assignments
        responsibility_data = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
```

```
)
consolidated_entities = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
    entities=responsibility_data
)
responsible_entities = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
)
responsibility_clarity = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
    entities=consolidated_entities
)
# Semantic traceability via embeddings
semantic_chunks = self._execute_method(
    "PolicyAnalysisEmbedder", "process_document", context,
    document_text=document_text,
    document_metadata=document_metadata
)
pdq_query = self._execute_method(
    "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
    pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
)
semantic_hits = self._execute_method(
    "PolicyAnalysisEmbedder", "semantic_search", context,
    query=pdq_query,
    document_chunks=semantic_chunks or []
)
diversified_hits = self._execute_method(
    "PolicyAnalysisEmbedder", "_apply_mmr", context,
    ranked_results=semantic_hits or []
)
semantic_cube_stub = self._execute_method(
    "SemanticAnalyzer", "_empty_semantic_cube", context
)
domain_scores = self._execute_method(
    "SemanticAnalyzer", "_classify_policy_domain", context,
    segment=document_text
)
cross_cutting = self._execute_method(
    "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
    segment=document_text
)
value_chain = self._execute_method(
    "SemanticAnalyzer", "_classify_value_chain_link", context,
    segment=document_text
)
semantic_vectors = self._execute_method(
    "SemanticAnalyzer", "_vectorize_segments", context,
    segments=[document_text]
)
processed_segment = self._execute_method(
    "SemanticAnalyzer", "_process_segment", context,
    segment=document_text,
    idx=0,
    vector=semantic_vectors[0] if semantic_vectors else None
)
semantic_complexity = self._execute_method(
    "SemanticAnalyzer", "_calculate_semantic_complexity", context,
    semantic_cube=semantic_cube_stub
)
evidence_confidence = self._execute_method(
    "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
    matches=[m.get("bpin","") for m in program_matches if isinstance(m, dict)],
    text_length=len(document_text),
    pattern_specificity=0.5
)

# Memory-safe entity processing with bounds checking
```

```
        MAX_ENTITY_SIZE = 1024 * 1024  # 1MB limit per entity
        entity_dicts = []
        for e in consolidated_entities[:5]:  # Already limited to 5 entities
            if not (isinstance(e, dict) or hasattr(e, "__dict__")):
                continue

            entity_size = sys.getsizeof(e)
            if entity_size > MAX_ENTITY_SIZE:
                logger.warning(f"Entity too large: {entity_size} bytes, skipping")
                continue

            try:
                entity_dict = self._execute_method(
                    "PDETMunicipalPlanAnalyzer", "_entity_to_dict", context, entity=e
                )
                entity_dicts.append(entity_dict)
            except MemoryError:
                logger.error("Memory exhausted during entity conversion, stopping")
                break
            except ExecutorFailure as e:
                logger.warning(f"Entity conversion failed: {e}")
                continue

        # Step 3: Generate traceability records
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            matches=program_matches
        )

        # Step 4: Generate PDQ report
        pdq_report = self._execute_method(
            "PolicyAnalysisEmbedder", "generate_pdq_report", context,
            traceability=traceability_record
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            entities=consolidated_entities
        )

        raw_evidence = {
            "budgetary_traceability": {
                "bpin_codes": [m.get("bpin") for m in (program_matches or []) if
m.get("bpin")],
                "project_codes": [m.get("project_code") for m in (program_matches or [])
if m.get("project_code")],
                "budget_matches": goal_budget_matches
            },
            "organizational_traceability": {
                "responsible_entities": consolidated_entities,
                "office_assignments": [e for e in (consolidated_entities or []) if
e.get("office")],
                "secretariat_assignments": [e for e in (consolidated_entities or []) if
e.get("secretariat")]
            },
            "traceability_record": traceability_record,
            "pdq_report": pdq_report,
            "accountability_matrix": accountability_matrix,
            "responsible_entities": responsible_entities,
            "responsibility_clarity_score": responsibility_clarity,
            "semantic_traceability": {
                "query": pdq_query,
                "semantic_hits": semantic_hits,
                "diversified_hits": diversified_hits
            },
            "semantic_cube_baseline": semantic_cube_stub,
            "policy_domain_scores": domain_scores,
```

```
            "responsibility_entities_dict": entity_dicts,
            "cross_cutting_themes": cross_cutting,
            "value_chain_links": value_chain,
            "semantic_vectors": semantic_vectors,
            "semantic_complexity": semantic_complexity,
            "evidence_confidence": evidence_confidence,
            "processed_segment": processed_segment
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "products_with_bpin": len([m for m in program_matches if isinstance(m,
dict) and m.get("bpin")]),
                "products_with_responsible": len(consolidated_entities) if
consolidated_entities else 0,
                "total_semantic_hits": len(semantic_hits) if semantic_hits else 0,
                "has_semantic_hits": bool(semantic_hits),
                "total_responsible_entities": len(responsible_entities) if
responsible_entities else 0,
                "has_responsible_entities": bool(responsible_entities),
                "total_diversified_hits": len(diversified_hits) if diversified_hits else
0,
                "total_entity_dicts": len(entity_dicts) if entity_dicts else 0,
                "has_semantic_vectors": bool(semantic_vectors),
                "total_semantic_vectors": len(semantic_vectors) if semantic_vectors else
0,
                "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
    """
    DIM03_Q04_TECHNICAL_FEASIBILITY — Evaluates activity-product feasibility vs
resources/deadlines (canonical D3).
    Epistemic mix: structural DAG validity, causal necessity, performance/implementation
readiness, and statistical robustness.

    Methods executed (in order):
    Step 1: Acyclicity p-value - AdvancedDAGValidator.calculate_acyclicity_pvalue
    Step 2: Acyclicity check - AdvancedDAGValidator._is_acyclic
    Step 3: Graph statistics - AdvancedDAGValidator.get_graph_stats
    Step 4: Node importance - AdvancedDAGValidator._calculate_node_importance
    Step 5: Subgraph generation - AdvancedDAGValidator._generate_subgraph
    Step 6: Node addition - AdvancedDAGValidator.add_node
    Step 7: Edge addition - AdvancedDAGValidator.add_edge
    Step 8: Node export - AdvancedDAGValidator.export_nodes
    Step 9: RNG initialization - AdvancedDAGValidator._initialize_rng
    Step 10: Statistical power - AdvancedDAGValidator._calculate_statistical_power
    Step 11: Node validator - AdvancedDAGValidator._get_node_validator
    Step 12: Empty result creation - AdvancedDAGValidator._create_empty_result
    Step 13: Necessity test - BayesianMechanismInference._test_necessity
    Step 14: Validation suite - IndustrialGradeValidator.execute_suite
    Step 15: Connection matrix - IndustrialGradeValidator.validate_connection_matrix
    Step 16: Performance benchmarks - IndustrialGradeValidator.run_performance_benchmarks
    Step 17: Benchmark operation - IndustrialGradeValidator._benchmark_operation
    Step 18: Metric logging - IndustrialGradeValidator._log_metric
    Step 19: Engine readiness - IndustrialGradeValidator.validate_engine_readiness
    Step 20: Performance analysis - PerformanceAnalyzer.analyze_performance
```

```
    Step 21: Loss functions - PerformanceAnalyzer._calculate_loss_functions
    Step 22: Resource likelihood - HierarchicalGenerativeModel._calculate_likelihood
    Step 23: ESS calculation - HierarchicalGenerativeModel._calculate_ess
    Step 24: R-hat calculation - HierarchicalGenerativeModel._calculate_r_hat
    Step 25: Causal categories validation -
IndustrialGradeValidator.validate_causal_categories
    Step 26: Category extraction - TeoriaCambio._extraer_categorias
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")

        # Step 1: Validate DAG structure
        acyclicity_pvalue = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )
        is_acyclic = self._execute_method(
            "AdvancedDAGValidator", "_is_acyclic", context
        )
        graph_stats = self._execute_method(
            "AdvancedDAGValidator", "get_graph_stats", context
        )
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )
        subgraph = self._execute_method(
            "AdvancedDAGValidator", "_generate_subgraph", context
        )
        added_node = self._execute_method(
            "AdvancedDAGValidator", "add_node", context,
            node_name="temp_node"
        )
        added_edge = self._execute_method(
            "AdvancedDAGValidator", "add_edge", context,
            source="temp_node",
            target="temp_target",
            weight=1.0
        )
        node_export = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context
        )
        rng_seed = self._execute_method(
            "AdvancedDAGValidator", "_initialize_rng", context,
            plan_name=plan_name,
            salt=dim_info.code
        )
        stat_power = self._execute_method(
            "AdvancedDAGValidator", "_calculate_statistical_power", context,
            s=int(graph_stats.get("edges", 0)),
            n=max(1, int(graph_stats.get("nodes", 1)))
        )
        node_validator = self._execute_method(
            "AdvancedDAGValidator", "_get_node_validator", context,
            node_type="producto"
        )
        empty_result = self._execute_method(
            "AdvancedDAGValidator", "_create_empty_result", context,
            plan_name=plan_name,
            seed=rng_seed,
            timestamp=context.get("metadata", {}).get("timestamp", "")
        )

        # Step 2: Test necessity of activities for products
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )
```

```
        # Step 3: Execute industrial-grade validation
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )
        performance_benchmarks = self._execute_method(
            "IndustrialGradeValidator", "run_performance_benchmarks", context
        )
        benchmark_ops = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )
        metric_log = self._execute_method(
            "IndustrialGradeValidator", "_log_metric", context,
            name="custom_latency",
            value=graph_stats.get("edges", 0),
            unit="edges",
            threshold=10.0
        )
        engine_readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 4: Analyze performance
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )
        loss_functions = self._execute_method(
            "PerformanceAnalyzer", "_calculate_loss_functions", context
        )
        # Likelihood estimation for resource adequacy
        resource_likelihood = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_likelihood", context,
            mechanism_type="tecnico",
            observations={"coherence": (performance_analysis or {}).get("resource_fit",
{}).get("score", 0.0)}
        )

        # Step 5: Calculate effective sample size
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context,
            chains=[]
        )
        causal_categories_valid = self._execute_method(
            "IndustrialGradeValidator", "validate_causal_categories", context
        )
        extracted_categories = self._execute_method(
            "TeoriaCambio", "_extraer_categorias", context,
            text=context.get("document_text", "")
        )

        raw_evidence = {
            "activity_product_mapping": connection_validation,
            "resource_adequacy": (performance_analysis or {}).get("resource_fit", {}),
            "timeline_feasibility": (performance_analysis or
{}).get("timeline_feasibility", {}),
            "technical_validation": {
                "dag_valid": is_acyclic,
                "acyclicity_p": acyclicity_pvalue,
                "necessity_score": necessity_test,
                "graph_stats": graph_stats,
                "node_importance": node_importance,
                "subgraph_sample": subgraph,
```

```
            "added_node": added_node,
            "added_edge": added_edge,
            "node_validator": node_validator,
            "empty_result": empty_result,
            "node_export": node_export,
            "rng_seed": rng_seed,
            "statistical_power": stat_power
        },
        "performance_metrics": {
            "benchmarks": performance_benchmarks,
            "loss_functions": loss_functions,
            "ess": ess,
            "r_hat": r_hat,
            "resource_likelihood": resource_likelihood
        },
        "engine_readiness": engine_readiness,
        "feasibility_score": validation_suite.get("overall_score", 0),
        "causal_categories_valid": causal_categories_valid,
        "extracted_categories": extracted_categories,
        "metric_log": metric_log
    }

    return {
        "executor_id": self.executor_id,
        "raw_evidence": raw_evidence,
        "metadata": {
            "methods_executed": [log["method"] for log in self.execution_log],
            "dag_is_valid": is_acyclic,
            "feasibility_score": (validation_suite or {}).get("overall_score", 0) if
validation_suite else 0,
            "total_graph_nodes": (graph_stats or {}).get("nodes", 0) if graph_stats
else 0,
            "total_graph_edges": (graph_stats or {}).get("edges", 0) if graph_stats
else 0,
            "has_node_export": bool(node_export),
            "total_exported_nodes": len(node_export) if node_export else 0,
            "has_subgraph": bool(subgraph),
            "total_extracted_categories": len(extracted_categories) if
extracted_categories else 0,
            "has_extracted_categories": bool(extracted_categories),
            "statistical_power": stat_power if isinstance(stat_power, (int, float))
else 0.0,
            "has_engine_readiness": bool(engine_readiness),
            "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
            "dimension_code": dim_info.code,
            "dimension_label": dim_info.label
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):
    """
    DIM03_Q05_OUTPUT_OUTCOME_LINKAGE — Analyzes mechanisms linking outputs to outcomes
with canonical D3 labeling.
    Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect
estimation, and Bayesian mechanism inference.

Methods (from D3-Q5):
    - PDETMunicipalPlanAnalyzer._identify_confounders
    - PDETMunicipalPlanAnalyzer._effect_to_dict
    - PDETMunicipalPlanAnalyzer._scenario_to_dict
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - PDETMunicipalPlanAnalyzer._identify_causal_nodes
```

- BayesianCounterfactualAuditor._evaluate_factual
- BayesianCounterfactualAuditor._evaluate_counterfactual
- CausalExtractor._assess_financial_consistency
- BayesianMechanismInference._infer_activity_sequence
- BayesianMechanismInference._generate_necessity_remediation
- BayesianCounterfactualAuditor.refutation_and_sanity_checks
- IndustrialPolicyProcessor._load_questionnaire
- PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
- PDETMunicipalPlanAnalyzer.construct_causal_dag
- PDETMunicipalPlanAnalyzer.estimate_causal_effects
- PDETMunicipalPlanAnalyzer.generate_counterfactuals
- CausalExtractor._build_type_hierarchy
- CausalExtractor._check_structural_violation
- CausalExtractor._calculate_type_transition_prior
- CausalExtractor._calculate_textual_proximity
- TeoriaCambio._validar_orden_causal
- PDETMunicipalPlanAnalyzer._refine_edge_probabilities
- PolicyAnalysisEmbedder.compare_policy_interventions
- BayesianMechanismInference.infer_mechanisms
"""

```
def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}
    dim_info = get_dimension_info(CanonicalDimension.D3.value)

    # Step 0: Build causal backbone and effects
    financial_analysis = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
    )
    causal_dag = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
        financial_analysis=financial_analysis
    )
    causal_effects = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "estimate_causal_effects", context,
        dag=causal_dag,
        financial_analysis=financial_analysis
    )
    counterfactuals = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context,
        dag=causal_dag,
        causal_effects=causal_effects,
        financial_analysis=financial_analysis
    )
    simulated_intervention = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
        intervention={},
        dag=causal_dag,
        causal_effects=causal_effects,
        label="baseline"
    )
    causal_nodes = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
        text=context.get("document_text", ""),
        tables=context.get("tables", []),
        financial_analysis=financial_analysis
    )
    confounders = {}
    for effect in causal_effects:
        treatment = effect.treatment if hasattr(effect, "treatment") else None
        outcome = effect.outcome if hasattr(effect, "outcome") else None
        if treatment and outcome:
            confounders[(treatment, outcome)] = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
                treatment=treatment,
                outcome=outcome,
                dag=causal_dag
            )
```

```
        effect_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context,
effect=effect)
            for effect in causal_effects
        ]
        scenario_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict",
context, scenario=scenario)
            for scenario in counterfactuals
        ]
        causal_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_analysis, "quality_score":
getattr(causal_dag, 'graph', {})}
        )
        factual_eval = None
        counterfactual_eval = None
        if causal_effects:
            first_effect = causal_effects[0]
            target = getattr(first_effect, "outcome", None) or ""
            evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
            factual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_factual", context,
                target=target,
                evidence=evidence
            )
            counterfactual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
                target=target,
                intervention={"shift": 0.1}
            )
        # Only catch specific expected exceptions, let system exceptions propagate
        matched_node = None
        try:
            matched_node = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_match_text_to_node", context,
                text=context.get("document_text", "")[:200],
                nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
            )
        except (KeyError, ValueError, TypeError, AttributeError, ExecutorFailure) as e:
            logger.warning(f"Node matching failed: {type(e).__name__}: {e}")
            matched_node = None
        # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError)
propagate

        # Step 1: Build type hierarchy
        type_hierarchy = self._execute_method(
            "CausalExtractor", "_build_type_hierarchy", context
        )

        # Step 2: Check structural violations
        structural_violations = self._execute_method(
            "CausalExtractor", "_check_structural_violation", context,
            hierarchy=type_hierarchy
        )

        # Step 3: Calculate transition priors and proximity
        transition_priors = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            hierarchy=type_hierarchy
        )
        textual_proximity = self._execute_method(
            "CausalExtractor", "_calculate_textual_proximity", context
        )

        # Step 4: Validate causal order
        causal_order_validation = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
```

```
            hierarchy=type_hierarchy
        )

        # Step 5: Refine edge probabilities
        refined_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            priors=transition_priors
        )
        financial_consistency = None
        if refined_edges:
            first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
            source = first_edge.get("source") if isinstance(first_edge, dict) else ""
            target = first_edge.get("target") if isinstance(first_edge, dict) else ""
            financial_consistency = self._execute_method(
                "CausalExtractor", "_assess_financial_consistency", context,
                source=source or "",
                target=target or ""
            )

        # Step 6: Compare policy interventions
        intervention_comparison = self._execute_method(
            "PolicyAnalysisEmbedder", "compare_policy_interventions", context
        )

        # Step 7: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context,
            edges=refined_edges
        )
        mechanism_sample = next(iter(mechanisms.values()), {})
        activity_sequence = self._execute_method(
            "BayesianMechanismInference", "_infer_activity_sequence", context,
            observations=mechanism_sample.get("observations", {}),
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico":
1.0})
        )
        quantified_uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico":
1.0}),
            sequence_posterior=mechanism_sample.get("activity_sequence", {}),
            coherence_score=mechanism_sample.get("coherence_score", 0.0)
        )
        mechanism_observations = self._execute_method(
            "BayesianMechanismInference", "_extract_observations", context,
            node={"id": next(iter(mechanisms.keys()), "")},
            text=context.get("document_text", "")
        )
        necessity_remediation = self._execute_method(
            "BayesianMechanismInference", "_generate_necessity_remediation", context,
            node_id=next(iter(mechanisms.keys()), ""),
            missing_components=structural_violations
        )
        questionnaire_stub = self._execute_method(
            "IndustrialPolicyProcessor", "_load_questionnaire", context
        )
        # Only catch specific expected exceptions, let system exceptions propagate
        refutation_checks = None
        try:
            confounder_keys = list(confounders.keys())
            first_pair = confounder_keys[0] if confounder_keys else ("", "")
            refutation_checks = self._execute_method(
                "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
                dag=getattr(causal_dag, "graph", None),
                target=first_pair[1],
                treatment=first_pair[0],
                confounders=list(confounders.values())[0] if confounders else []
            )
```

```
        except (KeyError, ValueError, TypeError, AttributeError, IndexError,
ExecutorFailure) as e:
            logger.warning(f"Refutation checks failed: {type(e).__name__}: {e}")
            refutation_checks = None
        # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError)
propagate

        raw_evidence = {
            "output_outcome_links": refined_edges,
            "mechanism_explanation": mechanisms,
            "type_hierarchy": type_hierarchy,
            "causal_dag": causal_dag,
            "causal_effects": causal_effects,
            "counterfactuals": counterfactuals,
            "simulated_intervention": simulated_intervention,
            "causal_nodes": causal_nodes,
            "financial_analysis": financial_analysis,
            "causal_validity": {
                "structural_violations": structural_violations,
                "order_valid": causal_order_validation
            },
            "transition_probabilities": transition_priors,
            "textual_proximity": textual_proximity,
            "intervention_comparison": intervention_comparison,
            "confounders": confounders,
            "effect_dicts": effect_dicts,
            "scenario_dicts": scenario_dicts,
            "activity_sequence_sample": activity_sequence,
            "uncertainty_quantified": quantified_uncertainty,
            "mechanism_observations": mechanism_observations,
            "refutation_checks": refutation_checks,
            "necessity_remediation": necessity_remediation,
            "questionnaire_stub": questionnaire_stub,
            "causal_recommendations": causal_recommendations,
            "financial_consistency": financial_consistency,
            "factual_eval": factual_eval,
            "counterfactual_eval": counterfactual_eval,
            "matched_node": matched_node
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(mechanisms or {}),
                "violations_found": len(structural_violations or []),
                "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 4: RESULTS & OUTCOMES
# ============================================================================

class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
    """
    DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS — Validates outcome indicators (baseline,
target, horizon) with canonical D4 notation.
    Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical
performance signals, and indicator quality scoring.
```

```
Methods (from D4-Q1):
- PDETMunicipalPlanAnalyzer._extract_entities_syntax
- PDETMunicipalPlanAnalyzer._extract_entities_ner
- CausalExtractor._calculate_language_specificity
- CausalExtractor._calculate_composite_likelihood
- CausalExtractor._calculate_semantic_distance
- TemporalLogicVerifier._classify_temporal_type
- PDETMunicipalPlanAnalyzer._score_indicators
- PDETMunicipalPlanAnalyzer._find_outcome_mentions
- PDETMunicipalPlanAnalyzer._score_temporal_consistency
- CausalExtractor._extract_goals
- CausalExtractor._parse_goal_context
- CausalExtractor._classify_goal_type
- TemporalLogicVerifier._parse_temporal_marker
- TemporalLogicVerifier._extract_resources
- TemporalLogicVerifier._should_precede
- PerformanceAnalyzer.analyze_performance
- PerformanceAnalyzer._generate_recommendations
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}
    dim_info = get_dimension_info(CanonicalDimension.D4.value)

    # Step 1: Find outcome mentions
    outcome_mentions = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_find_outcome_mentions", context
    )
    entities_syntax = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context,
        text=context.get("document_text", "")
    )
    entities_ner = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context,
        text=context.get("document_text", "")
    )

    # Step 2: Score temporal consistency
    temporal_consistency = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_score_temporal_consistency", context,
        outcomes=outcome_mentions
    )

    # Step 3: Extract and classify goals
    goals = self._execute_method(
        "CausalExtractor", "_extract_goals", context
    )
    goal_contexts = self._execute_method(
        "CausalExtractor", "_parse_goal_context", context,
        goals=goals
    )
    goal_types = self._execute_method(
        "CausalExtractor", "_classify_goal_type", context,
        goals=goals
    )
    semantic_distance = 0.0
    if goal_types and outcome_mentions:
        semantic_distance = self._execute_method(
            "CausalExtractor", "_calculate_semantic_distance", context,
            source=str(goal_types[0]),
            target=str(outcome_mentions[0])
        )

    # Step 4: Parse temporal markers
    temporal_markers = self._execute_method(
        "TemporalLogicVerifier", "_parse_temporal_marker", context,
        contexts=goal_contexts
```

```
        )
        temporal_type = self._execute_method(
            "TemporalLogicVerifier", "_classify_temporal_type", context,
            marker=temporal_markers[0] if temporal_markers else ""
        )
        resources_mentioned = self._execute_method(
            "TemporalLogicVerifier", "_extract_resources", context,
            text=context.get("document_text", "")
        )
        precedence_check = self._execute_method(
            "TemporalLogicVerifier", "_should_precede", context,
            marker_a=temporal_markers[0] if temporal_markers else "",
            marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
        )

        # Step 5: Analyze performance
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context,
            outcomes=outcome_mentions
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        performance_recommendations = self._execute_method(
            "PerformanceAnalyzer", "_generate_recommendations", context,
            performance_analysis=performance_analysis
        )
        # Semantic certainty for goals
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context,
            keyword=goal_contexts[0] if goal_contexts else "",
            policy_area=context.get("policy_area")
        )
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            evidence={
                "semantic_distance": indicator_quality if isinstance(indicator_quality,
(int, float)) else 0.0,
                "textual_proximity": performance_analysis.get("coherence_score", 0.0) if
isinstance(performance_analysis, dict) else 0.0,
                "language_specificity": language_specificity,
                "temporal_coherence": temporal_consistency if
isinstance(temporal_consistency, (int, float)) else 0.0
            }
        )

        raw_evidence = {
            "outcome_indicators": outcome_mentions,
            "indicators_with_baseline": [o for o in outcome_mentions if
o.get("has_baseline")],
            "indicators_with_target": [o for o in outcome_mentions if
o.get("has_target")],
            "indicators_with_horizon": [o for o in outcome_mentions if
o.get("time_horizon")],
            "temporal_consistency_score": temporal_consistency,
            "goal_classifications": goal_types,
            "temporal_markers": temporal_markers,
            "performance_metrics": performance_analysis,
            "indicator_quality": indicator_quality,
            "performance_recommendations": performance_recommendations,
            "entities_syntax": entities_syntax,
            "entities_ner": entities_ner,
            "temporal_type": temporal_type,
            "language_specificity": language_specificity,
            "composite_likelihood": composite_likelihood,
            "goal_outcome_semantic_distance": semantic_distance,
            "resources_mentioned": resources_mentioned,
            "precedence_check": precedence_check
```

```
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_outcomes": len(outcome_mentions or []),
                "complete_indicators": len([o for o in outcome_mentions or []
                    if o.get("has_baseline") and o.get("has_target") and
o.get("time_horizon")]),
                "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q2_CausalChainValidator(BaseExecutor):
    """
    Validates explicit causal chain with assumptions and enabling conditions.

    Methods (from D4-Q2):
    - TeoriaCambio._encontrar_caminos_completos
    - TeoriaCambio.validacion_completa
    - CausalExtractor.extract_causal_hierarchy
    - HierarchicalGenerativeModel.verify_conditional_independence
    - HierarchicalGenerativeModel._generate_independence_tests
    - BayesianCounterfactualAuditor.construct_scm
    - AdvancedDAGValidator._perform_sensitivity_analysis_internal
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Find complete causal paths
        complete_paths = self._execute_method(
            "TeoriaCambio", "_encontrar_caminos_completos", context
        )

        # Step 2: Complete validation
        validation_results = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            paths=complete_paths
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Verify conditional independence
        independence_verification = self._execute_method(
            "HierarchicalGenerativeModel", "verify_conditional_independence", context,
            hierarchy=causal_hierarchy
        )
        independence_tests = self._execute_method(
            "HierarchicalGenerativeModel", "_generate_independence_tests", context,
            verification=independence_verification
        )

        # Step 5: Construct structural causal model
        scm = self._execute_method(
```

```
            "BayesianCounterfactualAuditor", "construct_scm", context,
            hierarchy=causal_hierarchy
        )

        # Step 6: Perform sensitivity analysis
        sensitivity_analysis = self._execute_method(
            "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
            scm=scm
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            analysis=sensitivity_analysis
        )

        raw_evidence = {
            "causal_chain": complete_paths,
            "key_assumptions": validation_results.get("assumptions", []),
            "enabling_conditions": validation_results.get("conditions", []),
            "external_factors": validation_results.get("external_factors", []),
            "causal_hierarchy": causal_hierarchy,
            "independence_tests": independence_tests,
            "structural_model": scm,
            "sensitivity": sensitivity_analysis,
            "evidential_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "complete_paths_found": len(complete_paths),
                "assumptions_identified": len(validation_results.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
    """
    Analyzes justification of result ambition based on investment/capacity/benchmarks.

    Methods (from D4-Q3):
    - PDETMunicipalPlanAnalyzer._get_prior_effect
    - PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - AdaptivePriorCalculator.sensitivity_analysis
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator._calculate_statistical_power
    - BayesianMechanismInference._aggregate_bayesian_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Get prior effect estimates
        prior_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
        )

        # Step 2: Estimate effect using Bayesian methods
        effect_estimate = self._execute_method(
```

```
    "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
    priors=prior_effects
)

# Step 3: Compute robustness
robustness = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
    estimate=effect_estimate
)

# Step 4: Sensitivity analysis
sensitivity = self._execute_method(
    "AdaptivePriorCalculator", "sensitivity_analysis", context,
    estimate=effect_estimate
)

# Step 5: Calculate convergence diagnostics
r_hat = self._execute_method(
    "HierarchicalGenerativeModel", "_calculate_r_hat", context
)
ess = self._execute_method(
    "HierarchicalGenerativeModel", "_calculate_ess", context
)

# Step 6: Calculate statistical power
statistical_power = self._execute_method(
    "AdvancedDAGValidator", "_calculate_statistical_power", context,
    effect=effect_estimate
)

# Step 7: Aggregate confidence
confidence_aggregate = self._execute_method(
    "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
    estimates=[effect_estimate, robustness, statistical_power]
)

raw_evidence = {
    "ambition_level": context.get("target_ambition", {}),
    "financial_investment": context.get("total_investment", 0),
    "institutional_capacity": context.get("capacity_score", 0),
    "comparative_benchmarks": prior_effects,
    "justification_analysis": {
        "effect_estimate": effect_estimate,
        "robustness": robustness,
        "sensitivity": sensitivity,
        "statistical_power": statistical_power
    },
    "convergence_diagnostics": {
        "r_hat": r_hat,
        "ess": ess
    },
    "overall_confidence": confidence_aggregate
}

return {
    "executor_id": self.executor_id,
    "raw_evidence": raw_evidence,
    "metadata": {
        "methods_executed": [log["method"] for log in self.execution_log],
        "ambition_justified": confidence_aggregate > 0.7,
        "statistical_power": statistical_power
    },
    "execution_metrics": {
        "methods_count": len(self.execution_log),
        "all_succeeded": all(log["success"] for log in self.execution_log)
    }
}
```

```
class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
    """
    Evaluates whether results address/resolve prioritized problems from diagnosis.

    Methods (from D4-Q4):
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._identify_affected_sections
    - PolicyContradictionDetector._generate_resolution_recommendations
    - OperationalizationAuditor._generate_optimal_remediations
    - OperationalizationAuditor._get_remediation_text
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - FinancialAuditor._detect_allocation_gaps
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate objective alignment
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )

        # Step 2: Identify affected sections
        affected_sections = self._execute_method(
            "PolicyContradictionDetector", "_identify_affected_sections", context,
            alignment=objective_alignment
        )

        # Step 3: Generate resolution recommendations
        resolutions = self._execute_method(
            "PolicyContradictionDetector", "_generate_resolution_recommendations",
context,
            sections=affected_sections
        )

        # Step 4: Generate optimal remediations
        remediations = self._execute_method(
            "OperationalizationAuditor", "_generate_optimal_remediations", context
        )
        remediation_text = self._execute_method(
            "OperationalizationAuditor", "_get_remediation_text", context,
            remediations=remediations
        )

        # Step 5: Aggregate risk and prioritize
        risk_priorities = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
        )

        # Step 6: Detect allocation gaps
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )

        raw_evidence = {
            "prioritized_problems": context.get("diagnosis_problems", []),
            "proposed_results": context.get("outcome_indicators", []),
            "problem_result_mapping": objective_alignment,
            "unaddressed_problems": [p for p in affected_sections if not
p.get("addressed")],
            "solvency_score": objective_alignment.get("score", 0),
            "resolution_recommendations": resolutions,
            "remediations": remediation_text,
            "risk_priorities": risk_priorities,
            "allocation_gaps": allocation_gaps
        }
```

```
        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "problems_addressed": len([p for p in affected_sections if
p.get("addressed")]),
                "problems_unaddressed": len([p for p in affected_sections if not
p.get("addressed")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
    """
    Validates alignment with superior frameworks (PND, SDGs).

    Methods (from D4-Q5):
    - PDETMunicipalPlanAnalyzer._score_pdet_alignment
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - CDAFFramework._validate_dnp_compliance
    - CDAFFramework._generate_dnp_report
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - AdaptivePriorCalculator.validate_quality_criteria
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score PDET alignment
        pdet_alignment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_pdet_alignment", context
        )

        # Step 2: Score causal coherence
        causal_coherence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 3: Validate DNP compliance
        dnp_compliance = self._execute_method(
            "CDAFFramework", "_validate_dnp_compliance", context
        )
        dnp_report = self._execute_method(
            "CDAFFramework", "_generate_dnp_report", context,
            compliance=dnp_compliance
        )

        # Step 4: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            alignment=pdet_alignment
        )

        raw_evidence = {
            "pnd_alignment": dnp_compliance,
            "sdg_alignment": context.get("sdg_mappings", []),
            "pdet_alignment": pdet_alignment,
            "alignment_declarations": (dnp_report or {}).get("declarations", []),
```

```
            "causal_coherence": causal_coherence,
            "causal_dimensions": causal_dimensions,
            "quality_validation": quality_validation,
            "alignment_score": (pdet_alignment.get("score", 0) +
                                (dnp_compliance or {}).get("score", 0)) / 2
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "pnd_aligned": (dnp_compliance or {}).get("is_compliant", False),
                "sdgs_referenced": len(context.get("sdg_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 5: IMPACTS
# ============================================================================

class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
    """
    Analyzes long-term impacts, transmission routes, and time lags.

    Methods (from D5-Q1):
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_scenario_narrative
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - TeoriaCambio._validar_orden_causal
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._generate_interventions
    - BayesianCounterfactualAuditor.construct_scm
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Generate counterfactuals
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context
        )

        # Step 2: Simulate interventions
        simulation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            counterfactuals=counterfactuals
        )

        # Step 3: Generate scenario narratives
        scenarios = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_scenario_narrative", context,
            simulation=simulation
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Validate causal order
        causal_order = self._execute_method(
```

```
            "TeoriaCambio", "_validar_orden_causal", context,
            mediators=mediators
        )

        # Step 6: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 7: Generate interventions
        interventions = self._execute_method(
            "TextMiningEngine", "_generate_interventions", context
        )

        # Step 8: Construct SCM
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            order=causal_order
        )

        raw_evidence = {
            "long_term_impacts": context.get("impact_indicators", []),
            "structural_transformations": scenarios,
            "transmission_routes": mediators,
            "expected_time_lags": temporal_coherence.get("time_lags", []),
            "counterfactual_analysis": counterfactuals,
            "simulation_results": simulation,
            "causal_order_validation": causal_order,
            "causal_pathways": scm,
            "intervention_scenarios": interventions
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "impacts_defined": len(context.get("impact_indicators", [])),
                "mediators_identified": len(mediators or [])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
    """
    DIM05_Q02_COMPOSITE_PROXY_VALIDITY — Validates composite indices/proxies for complex
impacts (canonical D5).
    Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative
reporting quality, and semantic consistency.

    Methods executed (in order):
    Step 1: Quality score calculation - PDETMunicipalPlanAnalyzer.calculate_quality_score
    Step 2: Score confidence estimation -
PDETMunicipalPlanAnalyzer._estimate_score_confidence
    Step 3: E-value computation - PDETMunicipalPlanAnalyzer._compute_e_value
    Step 4: Robustness computation - PDETMunicipalPlanAnalyzer._compute_robustness_value
    Step 5: Sensitivity interpretation - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    Step 6: Reporting quality - ReportingEngine._calculate_quality_score
    Step 7: Bayesian confidence aggregation -
BayesianMechanismInference._aggregate_bayesian_confidence
    Step 8: Numerical consistency evaluation -
PolicyAnalysisEmbedder.evaluate_policy_numerical_consistency
    Step 9: Embedder diagnostics - PolicyAnalysisEmbedder.get_diagnostics
    Step 10: Document processing - PolicyAnalysisEmbedder.process_document
```

```
    Step 11: PDQ query generation - PolicyAnalysisEmbedder._generate_query_from_pdq
    Step 12: PDQ filtering - PolicyAnalysisEmbedder._filter_by_pdq
    Step 13: Numerical value extraction - PolicyAnalysisEmbedder._extract_numerical_values
    Step 14: Text embedding - PolicyAnalysisEmbedder._embed_texts
    Step 15: Overall confidence computation -
PolicyAnalysisEmbedder._compute_overall_confidence
    Step 16: Sufficiency calculation - FinancialAuditor._calculate_sufficiency
    Step 17: Overall quality interpretation -
PDETMunicipalPlanAnalyzer._interpret_overall_quality
    Step 18: Risk prioritization -
BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    Step 19: Unicode normalization - PolicyTextProcessor.normalize_unicode
    Step 20: Sentence segmentation - PolicyTextProcessor.segment_into_sentences
    Step 21: Evidence confidence - IndustrialPolicyProcessor._compute_evidence_confidence
    Step 22: Average confidence - IndustrialPolicyProcessor._compute_avg_confidence
    Step 23: Quality serialization - PDETMunicipalPlanAnalyzer._quality_to_dict
    Step 24: Evidence bundle - IndustrialPolicyProcessor._construct_evidence_bundle
    Step 25: Pattern compilation - PolicyTextProcessor.compile_pattern
    Step 26: Contextual window extraction - PolicyTextProcessor.extract_contextual_window
    Step 27: Executive report generation -
PDETMunicipalPlanAnalyzer.generate_executive_report
    Step 28: Results export - IndustrialPolicyProcessor.export_results
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D5.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Calculate quality scores
        quality_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context
        )
        score_confidence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_score_confidence", context,
            score=quality_score
        )

        # Step 2: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            score=quality_score
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            score=quality_score
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            e_value=e_value,
            robustness=robustness
        )

        # Step 3: Calculate reporting quality score
        reporting_quality = self._execute_method(
            "ReportingEngine", "_calculate_quality_score", context
        )

        # Step 4: Aggregate Bayesian confidence
        bayesian_confidence = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            scores=[quality_score, reporting_quality]
        )

        # Step 5: Evaluate numerical consistency
        numerical_consistency = self._execute_method(
            "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context
```

```
        )
        embedder_diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context
        )
        processed_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        pdq_filter = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        filtered_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
            chunks=processed_chunks,
            pdq_filter=pdq_filter
        )
        numerical_values = self._execute_method(
            "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
            chunks=processed_chunks
        )
        embedded_texts = self._execute_method(
            "PolicyAnalysisEmbedder", "_embed_texts", context,
            texts=[c.get("content", "") for c in processed_chunks] if
isinstance(processed_chunks, list) else []
        )
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else
[],
            numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict)
else {"evidence_strength": "weak", "numerical_coherence": 0.0}
        )

        # Step 6: Calculate sufficiency
        sufficiency = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )
        overall_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_overall_quality", context,
            score=getattr(quality_score, "overall_score", quality_score)
        )
        risk_prioritization = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            omission_score=1 - quality_score.financial_feasibility if
hasattr(quality_score, "financial_feasibility") else 0.2,
            insufficiency_score=1 - (sufficiency or {}).get("coverage_ratio", 0.0),
            unnecessity_score=1 - (robustness if isinstance(robustness, (int, float)) else
 0.0),
            causal_effect=e_value,
            feasibility=quality_score.financial_feasibility if hasattr(quality_score,
"financial_feasibility") else 0.8,
            cost=1.0
        )
        normalized_text = self._execute_method(
            "PolicyTextProcessor", "normalize_unicode", context,
            text=document_text
        )
        segmented_sentences = self._execute_method(
            "PolicyTextProcessor", "segment_into_sentences", context,
            text=document_text
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=context.get("proxy_indicators", []),
            text_length=len(document_text),
            pattern_specificity=0.5
```

```
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D5": {"dimension_confidence": (bayesian_confidence or
{}).get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0}}
        )
        quality_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
            quality=quality_score
        )
        evidence_bundle = self._execute_method(
            "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
            dimension=None,
            category="composite",
            matches=context.get("proxy_indicators", []),
            positions=[],
            confidence=bayesian_confidence.get("numerical_coherence", 0.0) if
isinstance(bayesian_confidence, dict) else 0.0
        )
        compiled_pattern = self._execute_method(
            "PolicyTextProcessor", "compile_pattern", context,
            pattern_str=r"[A-Z]{2,}\\s+\\d+"
        )
        contextual_window = self._execute_method(
            "PolicyTextProcessor", "extract_contextual_window", context,
            text=document_text,
            match_position=0,
            window_size=200
        )
        exec_report = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
            analysis_results={"quality_score": quality_dict, "financial_analysis":
context.get("financial_analysis", {}) or {"total_budget": 0, "funding_sources": {},
"confidence": (0, 0)}}
        )
        export_result = self._execute_method(
            "IndustrialPolicyProcessor", "export_results", context,
            results={"quality": quality_dict, "robustness": robustness},
            output_path="output/composite_results.json"
        )

        raw_evidence = {
            "composite_indices": context.get("composite_indicators", []),
            "proxy_indicators": context.get("proxy_indicators", []),
            "validity_justification": score_confidence,
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "interpretation": sensitivity_interpretation
            },
            "quality_scores": {
                "overall": quality_score,
                "reporting": reporting_quality
            },
            "bayesian_confidence": bayesian_confidence,
            "numerical_consistency": numerical_consistency,
            "measurement_sufficiency": sufficiency,
            "embedder_diagnostics": embedder_diagnostics,
            "quality_interpretation": overall_interpretation,
            "pdq_filter": pdq_filter,
            "filtered_chunks": filtered_chunks,
            "numerical_values": numerical_values,
            "embedded_texts": embedded_texts,
            "overall_confidence": overall_confidence,
            "risk_prioritization": risk_prioritization,
            "normalized_text": normalized_text,
            "segmented_sentences": segmented_sentences,
            "evidence_confidence": evidence_confidence,
```

```
            "avg_confidence": avg_confidence,
            "quality_dict": quality_dict,
            "compiled_pattern": compiled_pattern,
            "contextual_window": contextual_window,
            "evidence_bundle": evidence_bundle,
            "executive_report": exec_report,
            "export_result": export_result
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "composite_indices_count": len(context.get("composite_indicators", [])),
                "total_proxy_indicators": len(context.get("proxy_indicators", [])),
                "has_proxy_indicators": bool(context.get("proxy_indicators")),
                "total_numerical_values": len(numerical_values) if numerical_values else
0,
                "has_numerical_values": bool(numerical_values),
                "total_filtered_chunks": len(filtered_chunks) if filtered_chunks else 0,
                "has_filtered_chunks": bool(filtered_chunks),
                "total_segmented_sentences": len(segmented_sentences) if
segmented_sentences else 0,
                "has_segmented_sentences": bool(segmented_sentences),
                "total_embedded_texts": len(embedded_texts) if embedded_texts else 0,
                "has_embedded_texts": bool(embedded_texts),
                "validity_score": score_confidence,
                "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
    """
    Analyzes proxy indicators for intangible impacts with validity documentation.

    Methods (from D5-Q3):
    - CausalExtractor._calculate_semantic_distance
    - SemanticAnalyzer.extract_semantic_cube
    - BayesianMechanismInference._quantify_uncertainty
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - PolicyAnalysisEmbedder.get_diagnostics
    - AdaptivePriorCalculator._perturb_evidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate semantic distance
        semantic_distance = self._execute_method(
            "CausalExtractor", "_calculate_semantic_distance", context
        )

        # Step 2: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        # Step 3: Quantify uncertainty
        uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
```

```
            semantic_data=semantic_cube
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Get diagnostics
        diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context,
            mediators=mediators
        )

        # Step 6: Perturb evidence for sensitivity
        perturbed_evidence = self._execute_method(
            "AdaptivePriorCalculator", "_perturb_evidence", context,
            diagnostics=diagnostics
        )

        raw_evidence = {
            "intangible_impacts": context.get("intangible_indicators", []),
            "proxy_indicators": context.get("proxy_mappings", []),
            "validity_documentation": diagnostics,
            "limitations_acknowledged": (diagnostics or {}).get("limitations", []),
            "semantic_relationships": semantic_cube,
            "semantic_distance": semantic_distance,
            "uncertainty_quantification": uncertainty,
            "sensitivity_to_proxies": perturbed_evidence
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "intangibles_count": len(context.get("intangible_indicators", [])),
                "proxies_defined": len(context.get("proxy_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
    """
    Evaluates systemic risks that could rupture causal mechanisms.

    Methods (from D5-Q4):
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - BayesianCounterfactualAuditor._test_effect_stability
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._break_cycles
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit systemic risks
        systemic_risks = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )
```

```
        # Step 2: Refutation and sanity checks
        refutation = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            risks=systemic_risks
        )

        # Step 3: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            refutation=refutation
        )

        # Step 4: Interpret risks
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risks=systemic_risks
        )

        # Step 5: Interpret sensitivity
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            stability=effect_stability
        )

        # Step 6: Break cycles if present
        cycle_breaks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_break_cycles", context
        )

        # Step 7: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=systemic_risks
        )

        raw_evidence = {
            "macroeconomic_risks": [r for r in systemic_risks or [] if r.get("type") ==
"macroeconomic"],
            "environmental_risks": [r for r in systemic_risks or [] if r.get("type") ==
"environmental"],
            "political_risks": [r for r in systemic_risks or [] if r.get("type") ==
"political"],
            "mechanism_rupture_potential": (risk_interpretation or
{}).get("rupture_probability", 0),
            "effect_stability": effect_stability,
            "refutation_results": refutation,
            "sensitivity_analysis": sensitivity,
            "sensitivity_interpretation": sensitivity_interpretation,
            "cycle_vulnerabilities": cycle_breaks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "systemic_risks_identified": len(systemic_risks or []),
                "high_risk_count": len([r for r in systemic_risks or [] if
r.get("severity") == "high"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
```

```
"""
Analyzes realism of impact ambition and potential unintended effects.

Methods (from D5-Q5):
- HierarchicalGenerativeModel.posterior_predictive_check
- HierarchicalGenerativeModel._ablation_analysis
- HierarchicalGenerativeModel._calculate_waic_difference
- AdaptivePriorCalculator._add_ood_noise
- AdaptivePriorCalculator.validate_quality_criteria
- PDETMunicipalPlanAnalyzer._compute_e_value
- PDETMunicipalPlanAnalyzer._compute_robustness_value
- BayesianMechanismInference._calculate_coherence_factor
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Posterior predictive check
    predictive_check = self._execute_method(
        "HierarchicalGenerativeModel", "posterior_predictive_check", context
    )

    # Step 2: Ablation analysis
    ablation = self._execute_method(
        "HierarchicalGenerativeModel", "_ablation_analysis", context,
        check=predictive_check
    )

    # Step 3: Calculate WAIC difference
    waic_diff = self._execute_method(
        "HierarchicalGenerativeModel", "_calculate_waic_difference", context,
        ablation=ablation
    )

    # Step 4: Add out-of-distribution noise
    ood_analysis = self._execute_method(
        "AdaptivePriorCalculator", "_add_ood_noise", context
    )

    # Step 5: Validate quality criteria
    quality_validation = self._execute_method(
        "AdaptivePriorCalculator", "validate_quality_criteria", context,
        ood=ood_analysis
    )

    # Step 6: Compute robustness metrics
    e_value = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_compute_e_value", context
    )
    robustness = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context
    )

    # Step 7: Calculate coherence factor
    coherence = self._execute_method(
        "BayesianMechanismInference", "_calculate_coherence_factor", context
    )

    raw_evidence = {
        "impact_ambition_level": context.get("declared_ambition", 0),
        "realism_assessment": (predictive_check or {}).get("realism_score", 0),
        "negative_side_effects": (ablation or {}).get("negative_effects", []),
        "limit_hypotheses": (quality_validation or {}).get("limits", []),
        "robustness_metrics": {
            "e_value": e_value,
            "robustness": robustness,
            "coherence": coherence
        },
```

```
            "predictive_validity": predictive_check,
            "ablation_results": ablation,
            "model_comparison": waic_diff,
            "ood_sensitivity": ood_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "realism_score": (predictive_check or {}).get("realism_score", 0),
                "side_effects_identified": len((ablation or {}).get("negative_effects",
[]))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 6: CAUSALITY & THEORY OF CHANGE
# ============================================================================

class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
    """
    Builds/validates explicit Theory of Change with diagram and assumptions.

    Methods (from D6-Q1):
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio.validacion_completa
    - TeoriaCambio.export_nodes
    - ReportingEngine.generate_causal_diagram
    - ReportingEngine.generate_causal_model_json
    - AdvancedDAGValidator.export_nodes
    - PDETMunicipalPlanAnalyzer.export_causal_network
    - CausalExtractor.extract_causal_hierarchy
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context
        )

        # Step 2: Complete validation
        validation = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            graph=causal_graph
        )

        # Step 3: Export nodes from Theory of Change
        toc_nodes = self._execute_method(
            "TeoriaCambio", "export_nodes", context,
            graph=causal_graph
        )

        # Step 4: Generate causal diagram
        diagram = self._execute_method(
            "ReportingEngine", "generate_causal_diagram", context,
            graph=causal_graph
        )

        # Step 5: Generate causal model JSON
```

```
        model_json = self._execute_method(
            "ReportingEngine", "generate_causal_model_json", context,
            graph=causal_graph
        )

        # Step 6: Export nodes from DAG validator
        dag_nodes = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context,
            graph=causal_graph
        )

        # Step 7: Export causal network
        network_export = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "export_causal_network", context,
            graph=causal_graph
        )

        # Step 8: Extract causal hierarchy
        hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        raw_evidence = {
            "toc_exists": bool(causal_graph),
            "toc_diagram": diagram,
            "toc_json": model_json,
            "causal_graph": causal_graph,
            "nodes": toc_nodes,
            "dag_nodes": dag_nodes,
            "causes_identified": (hierarchy or {}).get("causes", []),
            "mediators_identified": (hierarchy or {}).get("mediators", []),
            "assumptions": (validation or {}).get("assumptions", []),
            "network_structure": network_export,
            "validation_results": validation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "nodes_count": len(toc_nodes or []),
                "assumptions_count": len((validation or {}).get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
    """
    Validates logical proportionality: no leaps, intervention matches result scale.

    Methods (from D6-Q2):
    - BeachEvidentialTest.apply_test_logic
    - BayesianMechanismInference._test_necessity
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._calculate_coherence_factor
    - BayesianCounterfactualAuditor._test_effect_stability
    - IndustrialGradeValidator.validate_connection_matrix
    - PolicyAnalysisEmbedder._compute_overall_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
```

```
        # Step 1: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context
        )

        # Step 2: Test necessity
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Test sufficiency
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )

        # Step 4: Calculate coherence factor
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            necessity=necessity_test,
            sufficiency=sufficiency_test
        )

        # Step 5: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context
        )

        # Step 6: Validate connection matrix
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )

        # Step 7: Compute overall confidence
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            tests=[necessity_test, sufficiency_test, effect_stability]
        )

        raw_evidence = {
            "logical_leaps_detected": (evidential_tests or {}).get("leaps", []),
            "intervention_scale": context.get("intervention_magnitude", 0),
            "result_scale": context.get("result_magnitude", 0),
            "proportionality_ratio": context.get("intervention_magnitude", 0) /
max(context.get("result_magnitude", 1), 1),
            "necessity_score": necessity_test,
            "sufficiency_score": sufficiency_test,
            "coherence_factor": coherence_factor,
            "effect_stability": effect_stability,
            "connection_validation": connection_validation,
            "overall_confidence": overall_confidence,
            "implementation_miracles": [leap for leap in (evidential_tests or
{}).get("leaps", [])
                                        if isinstance(leap, dict) and leap.get("type") ==
"miracle"]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "leaps_detected": len((evidential_tests or {}).get("leaps", [])),
                "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] -
1.0) < 0.5
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
```

```
        }
    }


class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
    """
    Analyzes validation/testing proposals for weak assumptions before scaling.

    Methods (from D6-Q3):
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_engine_readiness
    - IndustrialGradeValidator._benchmark_operation
    - AdaptivePriorCalculator.validate_quality_criteria
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Execute validation suite
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )

        # Step 2: Validate engine readiness
        readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 3: Benchmark operations
        benchmarks = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )

        # Step 4: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context
        )

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate acyclicity p-value
        acyclicity_p = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )

        # Step 7: Analyze performance
        performance = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "inconsistencies_recognized": (validation_suite or {}).get("inconsistencies",
[]),
            "weak_assumptions": (quality_validation or {}).get("weak_assumptions", []),
            "pilot_proposals": context.get("pilot_programs", []),
            "testing_proposals": context.get("testing_plans", []),
            "validation_before_scaling": (readiness or {}).get("ready_to_scale", False),
            "validation_results": validation_suite,
```

```
            "quality_criteria": quality_validation,
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess,
                "acyclicity_p": acyclicity_p
            },
            "performance_analysis": performance,
            "benchmarks": benchmarks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "inconsistencies_count": len((validation_suite or
{}).get("inconsistencies", [])),
                "pilots_proposed": len(context.get("pilot_programs", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):
    """
    Analyzes monitoring system with correction mechanisms and learning processes.

    Methods (from D6-Q4):
    - ConfigLoader.update_priors_from_feedback
    - ConfigLoader.check_uncertainty_reduction_criterion
    - ConfigLoader._save_prior_history
    - ConfigLoader._load_uncertainty_history
    - CDAFFramework._extract_feedback_from_audit
    - AdvancedDAGValidator._calculate_node_importance
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Update priors from feedback
        prior_updates = self._execute_method(
            "ConfigLoader", "update_priors_from_feedback", context
        )

        # Step 2: Check uncertainty reduction
        uncertainty_reduction = self._execute_method(
            "ConfigLoader", "check_uncertainty_reduction_criterion", context,
            updates=prior_updates
        )

        # Step 3: Save prior history
        history_saved = self._execute_method(
            "ConfigLoader", "_save_prior_history", context,
            updates=prior_updates
        )

        # Step 4: Load uncertainty history
        uncertainty_history = self._execute_method(
            "ConfigLoader", "_load_uncertainty_history", context
        )

        # Step 5: Extract feedback from audit
        feedback_extracted = self._execute_method(
            "CDAFFramework", "_extract_feedback_from_audit", context
```

```
        )

        # Step 6: Calculate node importance
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            updates=prior_updates
        )

        raw_evidence = {
            "monitoring_system_described": len(context.get("monitoring_indicators", [])) >
 0,
            "correction_mechanisms": (feedback_extracted or {}).get("mechanisms", []),
            "feedback_loops": (feedback_extracted or {}).get("loops", []),
            "learning_processes": (feedback_extracted or {}).get("learning", []),
            "prior_updates": prior_updates,
            "uncertainty_reduction": uncertainty_reduction,
            "history_saved": history_saved,
            "uncertainty_history": uncertainty_history,
            "node_importance": node_importance,
            "learning_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "feedback_mechanisms": len((feedback_extracted or {}).get("mechanisms",
[])),
                "learning_processes": len((feedback_extracted or {}).get("learning", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
    """
    Evaluates contextual adaptation: differential impacts and territorial constraints.

    Methods executed (in order):
    Step 1: Language specificity - CausalExtractor._calculate_language_specificity
    Step 2: Temporal coherence - CausalExtractor._assess_temporal_coherence
    Step 3: Critical links diagnosis - TextMiningEngine.diagnose_critical_links
    Step 4: Failure points identification - CausalInferenceSetup.identify_failure_points
    Step 5: Dynamics pattern - CausalInferenceSetup._get_dynamics_pattern
    Step 6: Text chunking - SemanticProcessor.chunk_text
    Step 7: PDM structure detection - SemanticProcessor._detect_pdm_structure
    Step 8: Table detection - SemanticProcessor._detect_table
    Step 9: Traceability record - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate language specificity
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context
        )

        # Step 2: Assess temporal coherence
```

```
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 3: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 4: Identify failure points
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 5: Get dynamics pattern
        dynamics_pattern = self._execute_method(
            "CausalInferenceSetup", "_get_dynamics_pattern", context
        )

        # Step 6: Process text structure
        text_chunks = self._execute_method(
            "SemanticProcessor", "chunk_text", context
        )
        pdm_structure = self._execute_method(
            "SemanticProcessor", "_detect_pdm_structure", context,
            chunks=text_chunks
        )
        table_detection = self._execute_method(
            "SemanticProcessor", "_detect_table", context,
            chunks=text_chunks
        )

        # Step 7: Generate traceability record
        traceability = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            specificity=language_specificity
        )

        raw_evidence = {
            "context_adaptation": (language_specificity or {}).get("adaptation_level", 0),
            "differential_impacts_recognized": (critical_links or
{}).get("differential_groups", []),
            "specific_groups_mentioned": (critical_links or {}).get("target_groups", []),
            "territorial_constraints": (failure_points or {}).get("territorial", []),
            "local_context_integration": (pdm_structure or {}).get("local_sections", []),
            "language_specificity": language_specificity,
            "temporal_coherence": temporal_coherence,
            "critical_links": critical_links,
            "failure_points": failure_points,
            "dynamics_pattern": dynamics_pattern,
            "structure_analysis": pdm_structure,
            "table_detection": table_detection,
            "text_chunks": text_chunks,
            "traceability": traceability
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "groups_identified": len((critical_links or {}).get("target_groups", []))
if critical_links else 0,
                "territorial_constraints": len((failure_points or {}).get("territorial",
[])) if failure_points else 0,
                "total_text_chunks": len(text_chunks) if text_chunks else 0,
                "has_text_chunks": bool(text_chunks),
                "total_table_detections": len(table_detection) if
```

```
isinstance(table_detection, list) else (1 if table_detection else 0),
                "has_table_detection": bool(table_detection),
                "has_pdm_structure": bool(pdm_structure),
                "has_dynamics_pattern": bool(dynamics_pattern),
                "has_traceability": bool(traceability)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# EXECUTOR REGISTRY
# ============================================================================

EXECUTOR_REGISTRY = {
    "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
    "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
    "D1-Q3": D1_Q3_BudgetAllocationTracer,
    "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
    "D1-Q5": D1_Q5_ScopeJustificationValidator,

    "D2-Q1": D2_Q1_StructuredPlanningValidator,
    "D2-Q2": D2_Q2_InterventionLogicInferencer,
    "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
    "D2-Q4": D2_Q4_RiskManagementAnalyzer,
    "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,

    "D3-Q1": D3_Q1_IndicatorQualityValidator,
    "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
    "D3-Q3": D3_Q3_TraceabilityValidator,
    "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
    "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,

    "D4-Q1": D4_Q1_OutcomeMetricsValidator,
    "D4-Q2": D4_Q2_CausalChainValidator,
    "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
    "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
    "D4-Q5": D4_Q5_VerticalAlignmentValidator,

    "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
    "D5-Q2": D5_Q2_CompositeMeasurementValidator,
    "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
    "D5-Q4": D5_Q4_SystemicRiskEvaluator,
    "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,

    "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
    "D6-Q2": D6_Q2_LogicalProportionalityValidator,
    "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
    "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
    "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
}


# ============================================================================
# PHASE 2 ORCHESTRATION
# ============================================================================


def _build_method_executor() -> MethodExecutor:
    """Construct a canonical MethodExecutor via the factory wiring."""
    bundle = build_processor()
    method_executor = getattr(bundle, "method_executor", None)
    if not isinstance(method_executor, MethodExecutor):
        raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor
instance.")
```

```
    return method_executor


def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
    """Build canonical metadata block using canonical_notation."""
    metadata: Dict[str, Any] = {}
    try:
        dim_key = executor_id.split("-")[0]
        dim_info = get_dimension_info(dim_key)
        metadata["dimension_code"] = dim_info.code
        metadata["dimension_label"] = dim_info.label
    except (KeyError, ValueError, IndexError, AttributeError) as e:
        logger.warning(f"Failed to load canonical metadata for {executor_id}: {e}")
        # Continue with empty metadata rather than failing
    # Let critical system exceptions (KeyboardInterrupt, SystemExit) propagate

    if executor_id in CANONICAL_QUESTION_LABELS:
        metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
    return metadata


def run_phase2_executors(context_package: Dict[str, Any],
                         policy_areas: List[str]) -> Dict[str, Any]:
    """
    Phase 2 Entry Point: Runs all 30 executors for each policy area.

    Args:
        context_package: Canonical package with document data from Phase 1
        policy_areas: List of policy area identifiers to analyze

    Returns:
        Dict mapping policy_area -> executor_id -> raw_evidence
    """
    results = {}
    method_executor = _build_method_executor()

    for policy_area in policy_areas:
        print(f"\n{'='*80}")
        print(f"Processing Policy Area: {policy_area}")
        print(f"{'='*80}\n")

        # Prepare context for this policy area
        area_context = {
            **context_package,
            "policy_area": policy_area
        }

        # Execute all 30 executors
        area_results = {}
        for executor_id, executor_class in EXECUTOR_REGISTRY.items():
            print(f"Running {executor_id}: {executor_class.__name__}...")

            try:
                # Instantiate executor with config
                config = load_executor_config(executor_id)
                executor = executor_class(executor_id, config,
method_executor=method_executor)

                # Execute and collect results
                result = executor.execute(area_context)
                # Append canonical metadata consistently
                result_metadata = result.get("metadata", {})
                result_metadata.update(_canonical_metadata(executor_id))
                result["metadata"] = result_metadata
                area_results[executor_id] = result

                print(f"  ✓ Success: {len(result['metadata']['methods_executed'])} methods
 executed")
```

```
            except ExecutorFailure as e:
                print(f"  ✗ FAILED: {str(e)}")
                raise  # Re-raise to stop execution as per requirement

        results[policy_area] = area_results

    return results


def load_executor_config(executor_id: str) -> Dict[str, Any]:
    """
    Load executor configuration from JSON contract.

    Args:
        executor_id: Executor identifier (e.g., "D1-Q1")

    Returns:
        Configuration dictionary from JSON contract
    """
    import json
    from pathlib import Path

    config_path = Path(f"config/executor_contracts/{executor_id}.json")

    if not config_path.exists():
        raise FileNotFoundError(f"Executor config not found: {config_path}")

    with open(config_path, 'r', encoding='utf-8') as f:
        return json.load(f)


# ==============================================================================
# EXAMPLE USAGE
# ==============================================================================

if __name__ == "__main__":
    # Example context package from Phase 1
    context_package = {
        "document_path": "data/pdm_municipality_xyz.pdf",
        "document_text": "...",  # Full document text
        "tables": [],  # Extracted tables from Phase 1
        "embeddings": {},  # Precomputed embeddings
        "entities": [],  # Pre-extracted entities
        "metadata": {
            "municipality": "Municipality XYZ",
            "year": 2024,
            "pages": 150
        }
    }

    # Policy areas to analyze
    policy_areas = [
        "PA01",  # Education
        "PA02",  # Health
        "PA03",  # Infrastructure
        # ... up to 10+ policy areas
    ]

    # Run Phase 2
    try:
        results = run_phase2_executors(context_package, policy_areas)
        print("\n" + "="*80)
        print("PHASE 2 COMPLETED SUCCESSFULLY")
        print("="*80)
        print(f"Processed {len(policy_areas)} policy areas")
        print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")
        print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")
```

```
    except ExecutorFailure as e:
        print("\n" + "="*80)
        print("PHASE 2 FAILED")
        print("="*80)
        print(f"Error: {str(e)}")
        print("Execution halted as per requirement: any method failure = executor
failure")

===== FILE: src/saaaaaa/core/orchestrator/executors_contract.py =====
from __future__ import annotations

from saaaaaa.core.orchestrator.base_executor_with_contract import BaseExecutorWithContract


class D1Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q1"


class D1Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q2"


class D1Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q3"


class D1Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q4"


class D1Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q5"


class D2Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q1"


class D2Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q2"


class D2Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q3"


class D2Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q4"
```

```
class D2Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q5"


class D3Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q1"


class D3Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q2"


class D3Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q3"


class D3Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q4"


class D3Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q5"


class D4Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q1"


class D4Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q2"


class D4Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q3"


class D4Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q4"


class D4Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q5"


class D5Q1_Executor_Contract(BaseExecutorWithContract):
```

```
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q1"


class D5Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q2"


class D5Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q3"


class D5Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q4"


class D5Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q5"


class D6Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q1"


class D6Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q2"


class D6Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q3"


class D6Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q4"


class D6Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q5"


# Aliases expected by core orchestrator
D1Q1_Executor = D1Q1_Executor_Contract
D1Q2_Executor = D1Q2_Executor_Contract
D1Q3_Executor = D1Q3_Executor_Contract
D1Q4_Executor = D1Q4_Executor_Contract
D1Q5_Executor = D1Q5_Executor_Contract
D2Q1_Executor = D2Q1_Executor_Contract
D2Q2_Executor = D2Q2_Executor_Contract
D2Q3_Executor = D2Q3_Executor_Contract
```

```
D2Q4_Executor = D2Q4_Executor_Contract
D2Q5_Executor = D2Q5_Executor_Contract
D3Q1_Executor = D3Q1_Executor_Contract
D3Q2_Executor = D3Q2_Executor_Contract
D3Q3_Executor = D3Q3_Executor_Contract
D3Q4_Executor = D3Q4_Executor_Contract
D3Q5_Executor = D3Q5_Executor_Contract
D4Q1_Executor = D4Q1_Executor_Contract
D4Q2_Executor = D4Q2_Executor_Contract
D4Q3_Executor = D4Q3_Executor_Contract
D4Q4_Executor = D4Q4_Executor_Contract
D4Q5_Executor = D4Q5_Executor_Contract
D5Q1_Executor = D5Q1_Executor_Contract
D5Q2_Executor = D5Q2_Executor_Contract
D5Q3_Executor = D5Q3_Executor_Contract
D5Q4_Executor = D5Q4_Executor_Contract
D5Q5_Executor = D5Q5_Executor_Contract
D6Q1_Executor = D6Q1_Executor_Contract
D6Q2_Executor = D6Q2_Executor_Contract
D6Q3_Executor = D6Q3_Executor_Contract
D6Q4_Executor = D6Q4_Executor_Contract
D6Q5_Executor = D6Q5_Executor_Contract

===== FILE: src/saaaaaa/core/orchestrator/executors_snapshot/executors.py =====
"""
executors.py - Phase 2: Executor Orchestration for Policy Document Analysis

This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate
methods from the core module to extract raw evidence from Colombian municipal
development plans (PDET/PDM documents).

Architecture:
- Each executor is independent and receives a canonical context package
- Methods execute in configured order; any failure causes executor failure
- Outputs are Python dicts/lists matching JSON contract specifications
- Executors are injected via MethodExecutor factory pattern

Usage:
    from factory import run_executor
    result = run_executor("D1-Q1", context_package)
"""

from typing import Dict, List, Any, Optional
from abc import ABC, abstractmethod

from saaaaaa.core.canonical_notation import CanonicalDimension, get_dimension_info
from saaaaaa.core.orchestrator.core import MethodExecutor
from saaaaaa.core.orchestrator.factory import build_processor

# Canonical question labels (only defined when verified in repo)
CANONICAL_QUESTION_LABELS = {
    "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
    "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
    "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
    "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
    "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
    "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
}

# Epistemic taxonomy per method (focused on executors expanded in this iteration)
EPISTEMIC_TAGS = {
    ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
    ("FinancialAuditor", "_match_program_to_node"): ["structural"],
    ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial",
"normative"],
    ("PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial",
"statistical"],
```

```
    ("PDETMunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
    ("PDETMunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic",
"structural"],
    ("PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical",
"causal"],
    ("PDETMunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
    ("PDETMunicipalPlanAnalyzer", "_identify_confounders"): ["causal", "consistency"],
    ("PDETMunicipalPlanAnalyzer", "_effect_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_scenario_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords"): ["semantic"],
    ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical",
"bayesian"],
    ("AdaptivePriorCalculator", "_adjust_domain_weights"): ["statistical"],
    ("BayesianMechanismInference", "_test_sufficiency"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_test_necessity"): ["statistical", "bayesian"],
    ("BayesianMechanismInference", "_log_refactored_components"): ["implementation"],
    ("BayesianMechanismInference", "_infer_activity_sequence"): ["causal"],
    ("BayesianMechanismInference", "infer_mechanisms"): ["causal", "bayesian"],
    ("AdvancedDAGValidator", "calculate_acyclicity_pvalue"): ["statistical",
"consistency"],
    ("AdvancedDAGValidator", "_is_acyclic"): ["structural", "consistency"],
    ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): ["statistical",
"bayesian"],
    ("AdvancedDAGValidator", "_calculate_confidence_interval"): ["statistical"],
    ("AdvancedDAGValidator", "_calculate_statistical_power"): ["statistical"],
    ("AdvancedDAGValidator", "_generate_subgraph"): ["structural"],
    ("AdvancedDAGValidator", "_get_node_validator"): ["implementation"],
    ("AdvancedDAGValidator", "_create_empty_result"): ["descriptive"],
    ("AdvancedDAGValidator", "_initialize_rng"): ["implementation"],
    ("AdvancedDAGValidator", "get_graph_stats"): ["structural"],
    ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
    ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
    ("AdvancedDAGValidator", "add_node"): ["structural"],
    ("AdvancedDAGValidator", "add_edge"): ["structural"],
    ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
    ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
    ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
    ("IndustrialGradeValidator", "_benchmark_operation"): ["implementation"],
    ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
    ("IndustrialGradeValidator", "_log_metric"): ["implementation"],
    ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
    ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_ess"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_likelihood"): ["statistical"],
    ("HierarchicalGenerativeModel", "_calculate_r_hat"): ["statistical"],
    ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
    ("ReportingEngine", "_calculate_quality_score"): ["normative", "statistical"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
    ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
    ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency",
"statistical"],
    ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
    ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_apply_mmr"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_generate_query_from_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_filter_by_pdq"): ["semantic"],
    ("PolicyAnalysisEmbedder", "_extract_numerical_values"): ["statistical"],
    ("PolicyAnalysisEmbedder", "_compute_overall_confidence"): ["statistical",
"normative"],
    ("PolicyAnalysisEmbedder", "_embed_texts"): ["semantic"],
```

```
    ("SemanticAnalyzer", "_classify_policy_domain"): ["semantic"],
    ("SemanticAnalyzer", "_empty_semantic_cube"): ["descriptive"],
    ("SemanticAnalyzer", "_classify_cross_cutting_themes"): ["semantic"],
    ("SemanticAnalyzer", "_classify_value_chain_link"): ["semantic"],
    ("SemanticAnalyzer", "_vectorize_segments"): ["semantic"],
    ("SemanticAnalyzer", "_calculate_semantic_complexity"): ["semantic"],
    ("SemanticAnalyzer", "_process_segment"): ["semantic"],
    ("PDETMunicipalPlanAnalyzer", "_entity_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_quality_to_dict"): ["descriptive", "normative"],
    ("PDETMunicipalPlanAnalyzer", "_deduplicate_tables"): ["structural",
"implementation"],
    ("PDETMunicipalPlanAnalyzer", "_indicator_to_dict"): ["descriptive"],
    ("PDETMunicipalPlanAnalyzer", "_generate_recommendations"): ["normative"],
    ("PDETMunicipalPlanAnalyzer", "_simulate_intervention"): ["causal", "statistical"],
    ("PDETMunicipalPlanAnalyzer", "_identify_causal_nodes"): ["structural", "causal"],
    ("PDETMunicipalPlanAnalyzer", "_match_text_to_node"): ["semantic", "structural"],
    ("TeoriaCambio", "_validar_orden_causal"): ["causal", "consistency"],
    ("TeoriaCambio", "_generar_sugerencias_internas"): ["normative"],
    ("TeoriaCambio", "_extraer_categorias"): ["semantic"],
    ("BayesianMechanismInference", "_extract_observations"): ["semantic", "causal"],
    ("BayesianMechanismInference", "_generate_necessity_remediation"): ["normative",
"causal"],
    ("BayesianMechanismInference", "_quantify_uncertainty"): ["statistical", "bayesian"],
    ("CausalExtractor", "_build_type_hierarchy"): ["structural"],
    ("CausalExtractor", "_check_structural_violation"): ["structural", "consistency"],
    ("CausalExtractor", "_calculate_type_transition_prior"): ["statistical", "bayesian"],
    ("CausalExtractor", "_calculate_textual_proximity"): ["semantic"],
    ("CausalExtractor", "_calculate_language_specificity"): ["semantic"],
    ("CausalExtractor", "_calculate_composite_likelihood"): ["statistical", "semantic"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
    ("CausalExtractor", "_extract_goals"): ["semantic"],
    ("CausalExtractor", "_parse_goal_context"): ["semantic"],
    ("CausalExtractor", "_classify_goal_type"): ["semantic"],
    ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
    ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
    ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
    ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural",
"semantic"],
    ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
    ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
    ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
    ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
    ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
    ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
    ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal",
"normative"],
    ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal",
"consistency"],
    ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
    ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal",
"statistical"],
    ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
    ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive",
"implementation"],
    ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural",
"semantic"],
    ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
    ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
    ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
    ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
    ("PDETMunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
    ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
}
```

```
class BaseExecutor(ABC):
    """
    Base class for all executors with standardized execution template.
    All executors must implement execute() and return structured evidence.
    """

    def __init__(self, executor_id: str, config: Dict[str, Any], method_executor:
MethodExecutor):
        self.executor_id = executor_id
        self.config = config
        if not isinstance(method_executor, MethodExecutor):
            raise RuntimeError("A valid MethodExecutor instance is required for executor
injection.")
        self.method_executor = method_executor
        self.execution_log = []
        self.dimension_info = None
        try:
            dim_key = executor_id.split("-")[0].replace("D", "D")
            self.dimension_info = get_dimension_info(dim_key)
        except Exception:
            self.dimension_info = None

    @abstractmethod
    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute configured methods and return raw evidence.

        Args:
            context: Canonical package with document, tables, metadata

        Returns:
            Dict with raw_evidence, metadata, execution_metrics

        Raises:
            ExecutorFailure: If any method fails
        """
        pass

    def _log_method_execution(self, class_name: str, method_name: str,
                              success: bool, result: Any = None, error: str = None):
        """Track method execution for debugging and traceability."""
        self.execution_log.append({
            "class": class_name,
            "method": method_name,
            "success": success,
            "result_type": type(result).__name__ if result else None,
            "error": error
        })

    def _execute_method(self, class_name: str, method_name: str,
                        context: Dict[str, Any], **kwargs) -> Any:
        """
        Execute a single method with error handling.

        Raises:
            ExecutorFailure: If method execution fails
        """
        try:
            # Method injection happens via factory - placeholder for actual execution
            method = self._get_method(class_name, method_name)
            result = method(context, **kwargs)
            self._log_method_execution(class_name, method_name, True, result)
            return result
        except Exception as e:
            self._log_method_execution(class_name, method_name, False, error=str(e))
            raise ExecutorFailure(
                f"Executor {self.executor_id} failed: {class_name}.{method_name} -
{str(e)}"
```

```
        )

    def _get_method(self, class_name: str, method_name: str):
        """Retrieve method using MethodExecutor to enforce routed execution."""
        if not isinstance(self.method_executor, MethodExecutor):
            raise RuntimeError(f"Invalid method executor provided:
{type(self.method_executor).__name__}")

        def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
            payload: Dict[str, Any] = {}
            if context:
                payload.update(context)
            if kwargs:
                payload.update(kwargs)
            return self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

        return _wrapped


class ExecutorFailure(Exception):
    """Raised when any method in an executor fails."""
    pass


# ============================================================================
# DIMENSION 1: DIAGNOSTICS & INPUTS
# ============================================================================

class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
    """
    Extracts numeric data, reference years, and official sources as baseline.

    Methods (from D1-Q1):
    - TextMiningEngine.diagnose_critical_links
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor.process
    - IndustrialPolicyProcessor._match_patterns_in_sentences
    - IndustrialPolicyProcessor._extract_point_evidence
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - FinancialAuditor._parse_amount
    - PDETMunicipalPlanAnalyzer._extract_financial_amounts
    - PDETMunicipalPlanAnalyzer._extract_from_budget_table
    - PolicyContradictionDetector._extract_quantitative_claims
    - PolicyContradictionDetector._parse_number
    - PolicyContradictionDetector._statistical_significance_test
    - BayesianNumericalAnalyzer.evaluate_policy_metric
    - BayesianNumericalAnalyzer.compare_policies
    - SemanticProcessor.chunk_text
    - SemanticProcessor.embed_single
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify critical data-bearing sections
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context,
            links=critical_links
        )
```

```
# Step 2: Extract structured quantitative claims
processed_sections = self._execute_method(
    "IndustrialPolicyProcessor", "process", context
)
pattern_matches = self._execute_method(
    "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,
    sections=processed_sections
)
point_evidence = self._execute_method(
    "IndustrialPolicyProcessor", "_extract_point_evidence", context,
    matches=pattern_matches
)

# Step 3: Parse numerical amounts and baseline data
parsed_amounts = self._execute_method(
    "FinancialAuditor", "_parse_amount", context,
    evidence=point_evidence
)
financial_amounts = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_extract_financial_amounts", context
)
budget_table_data = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_extract_from_budget_table", context
)

# Step 4: Extract temporal context (reference years)
goals = self._execute_method(
    "CausalExtractor", "_extract_goals", context
)
goal_contexts = self._execute_method(
    "CausalExtractor", "_parse_goal_context", context,
    goals=goals
)

# Step 5: Validate quantitative claims
quant_claims = self._execute_method(
    "PolicyContradictionDetector", "_extract_quantitative_claims", context
)
parsed_numbers = self._execute_method(
    "PolicyContradictionDetector", "_parse_number", context,
    claims=quant_claims
)
significance_test = self._execute_method(
    "PolicyContradictionDetector", "_statistical_significance_test", context,
    numbers=parsed_numbers
)

# Step 6: Evaluate baseline quality and compare
metric_evaluation = self._execute_method(
    "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
    metrics=parsed_numbers
)
policy_comparison = self._execute_method(
    "BayesianNumericalAnalyzer", "compare_policies", context,
    evaluations=metric_evaluation
)

# Step 7: Semantic validation of sources
text_chunks = self._execute_method(
    "SemanticProcessor", "chunk_text", context
)
embeddings = self._execute_method(
    "SemanticProcessor", "embed_single", context,
    chunks=text_chunks
)

# Assemble raw evidence
raw_evidence = {
```

```
            "numeric_data": parsed_numbers,
            "reference_years": [gc.get("year") for gc in goal_contexts if gc.get("year")],
            "official_sources": point_evidence.get("sources", []),
            "financial_baseline": financial_amounts,
            "budget_tables": budget_table_data,
            "significance_results": significance_test,
            "metric_evaluation": metric_evaluation,
            "source_embeddings": embeddings
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_numeric_claims": len(parsed_numbers),
                "sources_identified": len(point_evidence.get("sources", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
    """
    Quantifies problem magnitude, gaps, and identifies data limitations.

    Methods (from D1-Q2):
    - OperationalizationAuditor._audit_direct_evidence
    - OperationalizationAuditor._audit_systemic_risk
    - FinancialAuditor._detect_allocation_gaps
    - BayesianMechanismInference._detect_gaps
    - PDETMunicipalPlanAnalyzer._generate_optimal_remediations
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - BayesianCounterfactualAuditor.counterfactual_query
    - BayesianCounterfactualAuditor._test_effect_stability
    - PolicyContradictionDetector._detect_numerical_inconsistencies
    - PolicyContradictionDetector._calculate_numerical_divergence
    - BayesianConfidenceCalculator.calculate_posterior
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit evidence completeness
        direct_evidence_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_direct_evidence", context
        )
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 2: Detect gaps in resource allocation and mechanisms
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )
        mechanism_gaps = self._execute_method(
            "BayesianMechanismInference", "_detect_gaps", context
        )

        # Step 3: Generate optimal remediations and simulate interventions
        remediations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_optimal_remediations", context,
            gaps=allocation_gaps
        )
```

```
        simulation_results = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            remediations=remediations
        )

        # Step 4: Counterfactual analysis for problem dimensioning
        counterfactual = self._execute_method(
            "BayesianCounterfactualAuditor", "counterfactual_query", context
        )
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            counterfactual=counterfactual
        )

        # Step 5: Detect numerical inconsistencies
        numerical_inconsistencies = self._execute_method(
            "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context
        )
        divergence_calc = self._execute_method(
            "PolicyContradictionDetector", "_calculate_numerical_divergence", context,
            inconsistencies=numerical_inconsistencies
        )

        # Step 6: Calculate confidence and analyze performance
        posterior_confidence = self._execute_method(
            "BayesianConfidenceCalculator", "calculate_posterior", context,
            evidence=direct_evidence_audit
        )
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "magnitude_indicators": {
                "allocation_gaps": allocation_gaps,
                "mechanism_gaps": mechanism_gaps,
                "numerical_inconsistencies": numerical_inconsistencies
            },
            "deficit_quantification": divergence_calc,
            "data_limitations": {
                "evidence_gaps": direct_evidence_audit.get("gaps", []),
                "systemic_risks": systemic_risk_audit
            },
            "simulation_results": simulation_results,
            "confidence_scores": posterior_confidence,
            "performance_metrics": performance_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "gaps_identified": len(allocation_gaps) + len(mechanism_gaps),
                "inconsistencies_found": len(numerical_inconsistencies)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q3_BudgetAllocationTracer(BaseExecutor):
    """
    Traces monetary resources assigned to programs in Investment Plan (PPI).

    Methods (from D1-Q3):
```

```
- FinancialAuditor.trace_financial_allocation
- FinancialAuditor._process_financial_table
- FinancialAuditor._match_program_to_node
- FinancialAuditor._match_goal_to_budget
- FinancialAuditor._perform_counterfactual_budget_check
- FinancialAuditor._calculate_sufficiency
- PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
- PDETMunicipalPlanAnalyzer._extract_budget_for_pillar
- PDETMunicipalPlanAnalyzer._identify_funding_source
- PDETMunicipalPlanAnalyzer._classify_tables
- PDETMunicipalPlanAnalyzer._analyze_funding_sources
- PDETMunicipalPlanAnalyzer._score_financial_component
- BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Trace complete financial allocation chain
    allocation_trace = self._execute_method(
        "FinancialAuditor", "trace_financial_allocation", context
    )
    processed_tables = self._execute_method(
        "FinancialAuditor", "_process_financial_table", context
    )

    # Step 2: Match programs to budget nodes
    program_matches = self._execute_method(
        "FinancialAuditor", "_match_program_to_node", context,
        tables=processed_tables
    )
    goal_budget_matches = self._execute_method(
        "FinancialAuditor", "_match_goal_to_budget", context,
        programs=program_matches
    )

    # Step 3: Counterfactual checks and sufficiency calculation
    counterfactual_check = self._execute_method(
        "FinancialAuditor", "_perform_counterfactual_budget_check", context,
        matches=goal_budget_matches
    )
    sufficiency_calc = self._execute_method(
        "FinancialAuditor", "_calculate_sufficiency", context,
        allocation=allocation_trace
    )

    # Step 4: Analyze financial feasibility
    feasibility_analysis = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
    )
    pillar_budgets = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
    )
    funding_sources = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_identify_funding_source", context
    )

    # Step 5: Classify and analyze tables
    table_classification = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
        tables=processed_tables
    )
    funding_analysis = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
        sources=funding_sources
    )
    financial_score = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
```

```
            analysis=funding_analysis
        )

        # Step 6: Aggregate risk and prioritize
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            sufficiency=sufficiency_calc
        )

        raw_evidence = {
            "budget_allocations": allocation_trace,
            "program_mappings": program_matches,
            "goal_budget_links": goal_budget_matches,
            "sufficiency_analysis": sufficiency_calc,
            "pillar_budgets": pillar_budgets,
            "funding_sources": funding_sources,
            "financial_feasibility": feasibility_analysis,
            "financial_score": financial_score,
            "risk_priorities": risk_aggregation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "programs_traced": len(program_matches),
                "funding_sources_identified": len(funding_sources)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
    """
    Identifies installed capacity (entities, staff, equipment) and limitations.

    Methods (from D1-Q4):
    - PDETMunicipalPlanAnalyzer.identify_responsible_entities
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._classify_entity_type
    - PDETMunicipalPlanAnalyzer._score_entity_specificity
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - MechanismPartExtractor.extract_entity_activity
    - MechanismPartExtractor._normalize_entity
    - MechanismPartExtractor._validate_entity_activity
    - MechanismPartExtractor._calculate_ea_confidence
    - OperationalizationAuditor.audit_evidence_traceability
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Identify all responsible entities
        entities_identified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
        )

        # Step 2: Extract entities using NER and syntax
        ner_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context
        )
        syntax_entities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context
```

```
        )

        # Step 3: Classify and score entities
        entity_types = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
            entities=ner_entities + syntax_entities
        )
        specificity_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
            entities=entity_types
        )
        consolidated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
            entities=entity_types
        )

        # Step 4: Extract entity-activity relationships
        entity_activities = self._execute_method(
            "MechanismPartExtractor", "extract_entity_activity", context,
            entities=consolidated
        )
        normalized = self._execute_method(
            "MechanismPartExtractor", "_normalize_entity", context,
            activities=entity_activities
        )
        validated = self._execute_method(
            "MechanismPartExtractor", "_validate_entity_activity", context,
            normalized=normalized
        )
        ea_confidence = self._execute_method(
            "MechanismPartExtractor", "_calculate_ea_confidence", context,
            validated=validated
        )

        # Step 5: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            entity_activities=validated
        )

        raw_evidence = {
            "entities_identified": consolidated,
            "entity_types": entity_types,
            "specificity_scores": specificity_scores,
            "entity_activities": validated,
            "activity_confidence": ea_confidence,
            "capacity_indicators": {
                "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
                "equipment_mentions": [e for e in consolidated if e.get("type") ==
"equipment"],
                "organizational_units": [e for e in consolidated if e.get("type") ==
"organization"]
            },
            "limitations_identified": traceability_audit.get("gaps", []),
            "traceability_audit": traceability_audit
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "entities_count": len(consolidated),
                "activities_extracted": len(validated)
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
```

```
        }
    }


class D1_Q5_ScopeJustificationValidator(BaseExecutor):
    """
    Validates scope justification via legal framework and constraint recognition.

    Methods (from D1-Q5):
    - TemporalLogicVerifier._check_deadline_constraints
    - TemporalLogicVerifier.verify_temporal_consistency
    - CausalInferenceSetup.identify_failure_points
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._analyze_link_text
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - IndustrialPolicyProcessor._extract_metadata
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Verify temporal constraints
        deadline_constraints = self._execute_method(
            "TemporalLogicVerifier", "_check_deadline_constraints", context
        )
        temporal_consistency = self._execute_method(
            "TemporalLogicVerifier", "verify_temporal_consistency", context
        )

        # Step 2: Identify failure points in scope
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 3: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 4: Analyze link text for justifications
        link_analysis = self._execute_method(
            "TextMiningEngine", "_analyze_link_text", context
        )

        # Step 5: Analyze causal dimensions and extract metadata
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )
        metadata_extracted = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context,
            dimensions=causal_dimensions
        )

        raw_evidence = {
            "legal_framework_citations": metadata_extracted.get("legal_refs", []),
            "temporal_constraints": {
                "deadline_checks": deadline_constraints,
                "consistency": temporal_consistency,
                "coherence": temporal_coherence
            },
            "budgetary_constraints": metadata_extracted.get("budget_limits", []),
            "competence_constraints": metadata_extracted.get("competence_refs", []),
            "failure_points": failure_points,
            "scope_justifications": link_analysis.get("justifications", []),
            "causal_dimensions": causal_dimensions
        }

        return {
```

```
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "constraints_identified": len(deadline_constraints),
                "legal_citations": len(metadata_extracted.get("legal_refs", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 2: ACTIVITY DESIGN
# ============================================================================

class D2_Q1_StructuredPlanningValidator(BaseExecutor):
    """
    Validates structured format of activities (table/matrix with required columns).

    Methods (from D2-Q1):
    - PDFProcessor.extract_tables
    - FinancialAuditor._process_financial_table
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._classify_tables
    - PDETMunicipalPlanAnalyzer._is_likely_header
    - PDETMunicipalPlanAnalyzer._clean_dataframe
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract all tables
        extracted_tables = self._execute_method(
            "PDFProcessor", "extract_tables", context
        )

        # Step 2: Process financial tables
        processed_tables = self._execute_method(
            "FinancialAuditor", "_process_financial_table", context,
            tables=extracted_tables
        )

        # Step 3: Deduplicate and classify tables
        deduplicated = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=processed_tables
        )
        classified = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
            tables=deduplicated
        )

        # Step 4: Identify headers and clean dataframes
        header_checks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_is_likely_header", context,
            tables=classified
        )
        cleaned = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_clean_dataframe", context,
            tables=classified
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
```

```
            "ReportingEngine", "generate_accountability_matrix", context,
            tables=cleaned
        )

        raw_evidence = {
            "tables_extracted": len(extracted_tables),
            "activity_tables": [t for t in classified if t.get("type") == "activity"],
            "matrix_structure": accountability_matrix,
            "required_columns_present": {
                "responsible_entity": any("responsible" in str(t.get("columns",
[])).lower()
                                          for t in cleaned),
                "deliverable": any("deliverable" in str(t.get("columns", [])).lower()
                                   for t in cleaned),
                "timeline": any("timeline" in str(t.get("columns", [])).lower()
                                for t in cleaned),
                "cost": any("cost" in str(t.get("columns", [])).lower()
                            for t in cleaned)
            },
            "table_quality": {
                "clean_tables": len(cleaned),
                "with_headers": sum(1 for h in header_checks if h)
            }
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_tables": len(extracted_tables),
                "activity_tables": len([t for t in classified if t.get("type") ==
"activity"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q2_InterventionLogicInferencer(BaseExecutor):
    """
    Infers intervention logic: instrument (how), target (who), causality (why).

    Methods (from D2-Q2):
    - BayesianMechanismInference.infer_mechanisms
    - BayesianMechanismInference._infer_single_mechanism
    - BayesianMechanismInference._infer_mechanism_type
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - CausalExtractor.extract_causal_hierarchy
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio._es_conexion_valida
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - BeachEvidentialTest.classify_test
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context
        )
        single_mechanisms = []
        for mech in mechanisms:
```

```
    single = self._execute_method(
        "BayesianMechanismInference", "_infer_single_mechanism", context,
        mechanism=mech
    )
    single_mechanisms.append(single)

mechanism_types = self._execute_method(
    "BayesianMechanismInference", "_infer_mechanism_type", context,
    mechanisms=single_mechanisms
)

# Step 2: Test sufficiency and necessity
sufficiency_tests = self._execute_method(
    "BayesianMechanismInference", "_test_sufficiency", context,
    mechanisms=single_mechanisms
)
necessity_tests = self._execute_method(
    "BayesianMechanismInference", "_test_necessity", context,
    mechanisms=single_mechanisms
)

# Step 3: Extract causal hierarchy
causal_hierarchy = self._execute_method(
    "CausalExtractor", "extract_causal_hierarchy", context
)

# Step 4: Build causal graph
causal_graph = self._execute_method(
    "TeoriaCambio", "construir_grafo_causal", context,
    hierarchy=causal_hierarchy
)
connection_validation = self._execute_method(
    "TeoriaCambio", "_es_conexion_valida", context,
    graph=causal_graph
)

# Step 5: Construct DAG
causal_dag = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
    graph=causal_graph
)

# Step 6: Classify evidential tests
evidential_tests = self._execute_method(
    "BeachEvidentialTest", "classify_test", context,
    mechanisms=single_mechanisms
)

# Step 7: Analyze causal dimensions
causal_dimensions = self._execute_method(
    "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
)

raw_evidence = {
    "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
    "target_populations": [m.get("target") for m in single_mechanisms],
    "causal_logic": {
        "mechanisms": single_mechanisms,
        "mechanism_types": mechanism_types,
        "sufficiency": sufficiency_tests,
        "necessity": necessity_tests
    },
    "causal_hierarchy": causal_hierarchy,
    "causal_graph": causal_graph,
    "causal_dag": causal_dag,
    "evidential_strength": evidential_tests,
    "dimensions": causal_dimensions
}
```

```
        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "mechanisms_identified": len(single_mechanisms),
                "instruments_found": len([m for m in single_mechanisms if
m.get("instrument")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
    """
    Analyzes linkage between activities and root causes/structural determinants.

    Methods (from D2-Q3):
    - CausalExtractor._extract_causal_links
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._initialize_prior
    - CausalExtractor._calculate_type_transition_prior
    - PDETMunicipalPlanAnalyzer._identify_causal_edges
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - BayesianCounterfactualAuditor.construct_scm
    - BayesianCounterfactualAuditor._create_default_equations
    - SemanticAnalyzer.extract_semantic_cube
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Extract causal links
        causal_links = self._execute_method(
            "CausalExtractor", "_extract_causal_links", context
        )

        # Step 2: Calculate likelihoods
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            links=causal_links
        )
        prior_init = self._execute_method(
            "CausalExtractor", "_initialize_prior", context
        )
        type_transition_prior = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            links=causal_links
        )

        # Step 3: Identify and refine causal edges
        causal_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_edges", context,
            links=causal_links
        )
        refined_probabilities = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            edges=causal_edges
        )

        # Step 4: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            edges=refined_probabilities
```

```
        )
        default_equations = self._execute_method(
            "BayesianCounterfactualAuditor", "_create_default_equations", context,
            scm=scm
        )

        # Step 5: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        raw_evidence = {
            "root_causes_identified": [link.get("root_cause") for link in causal_links],
            "activity_linkages": causal_links,
            "link_probabilities": refined_probabilities,
            "composite_likelihood": composite_likelihood,
            "structural_model": scm,
            "model_equations": default_equations,
            "semantic_relationships": semantic_cube,
            "determinants_addressed": [link for link in causal_links if
link.get("addresses_determinant")]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "causal_links_found": len(causal_links),
                "root_causes_count": len(set(link.get("root_cause") for link in
causal_links))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
    """
    Identifies implementation risks and mitigation measures.

    Methods (from D2-Q4):
    - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
    - PDETMunicipalPlanAnalyzer.sensitivity_analysis
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Bayesian risk inference
        risk_inference = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
        )

        # Step 2: Sensitivity analysis
        sensitivity = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
            risks=risk_inference
```

```
        )

        # Step 3: Risk interpretation
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            inference=risk_inference
        )

        # Step 4: Compute robustness metrics
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            sensitivity=sensitivity
        )
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            robustness=robustness
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            sensitivity=sensitivity
        )

        # Step 5: Audit systemic risks
        systemic_risk_audit = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )

        # Step 6: Aggregate and prioritize risks
        risk_aggregation = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            risks=risk_inference
        )

        # Step 7: Refutation and sanity checks
        refutation_checks = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            aggregation=risk_aggregation
        )

        # Step 8: Additional sensitivity analysis
        adaptive_sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=risk_inference
        )

        raw_evidence = {
            "operational_risks": [r for r in risk_inference if r.get("type") ==
"operational"],
            "social_risks": [r for r in risk_inference if r.get("type") == "social"],
            "security_risks": [r for r in risk_inference if r.get("type") == "security"],
            "mitigation_measures": risk_interpretation.get("mitigations", []),
            "risk_priorities": risk_aggregation,
            "robustness_metrics": {
                "robustness_value": robustness,
                "e_value": e_value
            },
            "sensitivity_analysis": sensitivity,
            "systemic_risks": systemic_risk_audit,
            "validation_checks": refutation_checks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "risks_identified": len(risk_inference),
                "mitigations_proposed": len(risk_interpretation.get("mitigations", []))
```

```
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
    """
    Evaluates strategic coherence: complementarity and logical sequence.

    Methods (from D2-Q5):
    - PolicyContradictionDetector._detect_logical_incompatibilities
    - PolicyContradictionDetector._calculate_coherence_metrics
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._calculate_graph_fragmentation
    - OperationalizationAuditor.audit_sequence_logic
    - BayesianMechanismInference._calculate_coherence_factor
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Detect logical incompatibilities
        incompatibilities = self._execute_method(
            "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
        )

        # Step 2: Calculate coherence metrics
        coherence_metrics = self._execute_method(
            "PolicyContradictionDetector", "_calculate_coherence_metrics", context
        )
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )
        graph_fragmentation = self._execute_method(
            "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
        )

        # Step 3: Audit sequence logic
        sequence_audit = self._execute_method(
            "OperationalizationAuditor", "audit_sequence_logic", context
        )

        # Step 4: Calculate coherence factors
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            metrics=coherence_metrics
        )
        causal_coherence_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 5: Adaptive likelihood calculation
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
            coherence=causal_coherence_score
        )

        raw_evidence = {
            "complementarity_evidence": coherence_metrics.get("complementarity", []),
            "sequential_logic": sequence_audit,
            "logical_incompatibilities": incompatibilities,
            "coherence_scores": {
                "overall_coherence": coherence_metrics,
```

```
            "objective_alignment": objective_alignment,
            "causal_coherence": causal_coherence_score,
            "coherence_factor": coherence_factor
        },
        "graph_metrics": {
            "fragmentation": graph_fragmentation
        },
        "adaptive_likelihood": adaptive_likelihood
    }

    return {
        "executor_id": self.executor_id,
        "raw_evidence": raw_evidence,
        "metadata": {
            "methods_executed": [log["method"] for log in self.execution_log],
            "incompatibilities_found": len(incompatibilities),
            "coherence_score": coherence_metrics.get("score", 0)
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


# ============================================================================
# DIMENSION 3: PRODUCTS & OUTPUTS
# ============================================================================

class D3_Q1_IndicatorQualityValidator(BaseExecutor):
    """
    Validates indicator quality: baseline, target, source of verification.

    Methods (from D3-Q1):
    - PDETMunicipalPlanAnalyzer._score_indicators
    - OperationalizationAuditor.audit_evidence_traceability
    - CausalInferenceSetup.assign_probative_value
    - BeachEvidentialTest.apply_test_logic
    - TextMiningEngine.diagnose_critical_links
    - IndustrialPolicyProcessor._extract_metadata
    - IndustrialPolicyProcessor._calculate_quality_score
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score indicators
        indicator_scores = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )

        # Step 2: Audit evidence traceability
        traceability_audit = self._execute_method(
            "OperationalizationAuditor", "audit_evidence_traceability", context,
            indicators=indicator_scores
        )

        # Step 3: Assign probative value
        probative_values = self._execute_method(
            "CausalInferenceSetup", "assign_probative_value", context,
            indicators=indicator_scores
        )

        # Step 4: Apply evidential tests
        evidential_tests = self._execute_method(
            "BeachEvidentialTest", "apply_test_logic", context,
            indicators=indicator_scores
```

```
        )

        # Step 5: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
        )

        # Step 6: Extract and score metadata
        metadata = self._execute_method(
            "IndustrialPolicyProcessor", "_extract_metadata", context
        )
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context,
            metadata=metadata
        )

        # Step 7: Generate traceability record
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            indicators=indicator_scores
        )

        raw_evidence = {
            "indicators_with_baseline": [i for i in indicator_scores if
i.get("has_baseline")],
            "indicators_with_target": [i for i in indicator_scores if
i.get("has_target")],
            "indicators_with_source": [i for i in indicator_scores if
i.get("has_source")],
            "indicator_quality_scores": indicator_scores,
            "traceability": traceability_audit,
            "probative_values": probative_values,
            "evidential_strength": evidential_tests,
            "overall_quality_score": quality_score,
            "traceability_record": traceability_record
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_indicators": len(indicator_scores),
                "complete_indicators": len([i for i in indicator_scores
                    if i.get("has_baseline") and i.get("has_target") and
i.get("has_source")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
    """
    DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY — Analyzes proportionality of targets to the
diagnosed universe using canonical D3 notation.
    Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes
 tests, and semantic indicator quality.

    Methods (from D3-Q2):
    - AdvancedDAGValidator._calculate_bayesian_posterior
    - AdvancedDAGValidator._calculate_confidence_interval
    - AdaptivePriorCalculator._adjust_domain_weights
    - PDETMunicipalPlanAnalyzer._get_spanish_stopwords
    - BayesianMechanismInference._log_refactored_components
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
```

```
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - FinancialAuditor._calculate_sufficiency
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._test_necessity
    - PDETMunicipalPlanAnalyzer._assess_financial_sustainability
    - AdaptivePriorCalculator.calculate_likelihood_adaptativo
    - IndustrialPolicyProcessor._calculate_quality_score
    - TeoriaCambio._generar_sugerencias_internas
    - PDETMunicipalPlanAnalyzer._deduplicate_tables
    - PDETMunicipalPlanAnalyzer._indicator_to_dict
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - IndustrialPolicyProcessor._compile_pattern_registry
    - IndustrialPolicyProcessor._build_point_patterns
    - IndustrialPolicyProcessor._empty_result
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Financial feasibility snapshot and indicator quality
        financial_feasibility = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        spanish_stopwords = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords", context
        )
        funding_sources = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
            financial_indicators=financial_feasibility.get("financial_indicators", []),
            tables=context.get("tables", [])
        )
        financial_component = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
            financial_analysis=financial_feasibility
        )
        pattern_registry = self._execute_method(
            "IndustrialPolicyProcessor", "_compile_pattern_registry", context
        )
        point_patterns = self._execute_method(
            "IndustrialPolicyProcessor", "_build_point_patterns", context
        )
        empty_policy_result = self._execute_method(
            "IndustrialPolicyProcessor", "_empty_result", context
        )
        dedup_tables = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
            tables=context.get("tables", [])
        )
        first_indicator = None
        if isinstance(financial_feasibility.get("financial_indicators", []), list):
            inds = financial_feasibility.get("financial_indicators", [])
            first_indicator = inds[0] if inds else None
        indicator_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
            ind=first_indicator if first_indicator else {}
        )
        proportionality_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_feasibility,
"quality_score": quality_score} if 'quality_score' in locals() else {}
        )

        # Step 1: Calculate sufficiency
```

```
        sufficiency_calc = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )

        # Step 2: Test sufficiency and necessity of targets
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Assess financial sustainability
        sustainability_assessment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability", context
        )
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)
        )

        # Step 4: Calculate adaptive likelihood
        adaptive_likelihood = self._execute_method(
            "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context
        )
        domain_scores = {
            "structural": sufficiency_calc.get("coverage_ratio", 0.0),
            "financial": financial_feasibility.get("sustainability_score", 0.0),
            "semantic": indicator_quality if isinstance(indicator_quality, (int, float))
else 0.0
        }
        adjusted_weights = self._execute_method(
            "AdaptivePriorCalculator", "_adjust_domain_weights", context,
            domain_scores=domain_scores
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D3": {"dimension_confidence":
domain_scores.get("structural", 0.0)}}
        )

        # Step 5: Calculate quality score
        quality_score = self._execute_method(
            "IndustrialPolicyProcessor", "_calculate_quality_score", context
        )

        # Step 6: Generate internal suggestions
        internal_suggestions = self._execute_method(
            "TeoriaCambio", "_generar_sugerencias_internas", context
        )
        # Bayesian posterior diagnostics for proportionality evidence
        posterior_probability = self._execute_method(
            "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,
            likelihood=sufficiency_calc.get("coverage_ratio", 0.5),
            prior=0.5
        )
        confidence_interval = self._execute_method(
            "AdvancedDAGValidator", "_calculate_confidence_interval", context,
            s=int(sufficiency_calc.get("covered_targets", 0)),
            n=max(1, int(sufficiency_calc.get("targets_total",
len(context.get("product_targets", [])))))),
            conf=0.95
        )
        self._execute_method(
            "BayesianMechanismInference", "_log_refactored_components", context
        )

        raw_evidence = {
```

```
            "target_population_size": context.get("diagnosed_universe", 0),
            "product_targets": context.get("product_targets", []),
            "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
            "dosage_analysis": sufficiency_calc.get("dosage", {}),
            "sufficiency_test": sufficiency_test,
            "necessity_test": necessity_test,
            "sustainability": sustainability_assessment,
            "financial_feasibility": financial_feasibility,
            "indicator_quality": indicator_quality,
            "risk_interpretation": risk_interpretation,
            "proportionality_score": quality_score,
            "recommendations": internal_suggestions,
            "stopwords_spanish": spanish_stopwords,
            "funding_sources_analysis": funding_sources,
            "financial_component_score": financial_component,
            "pattern_registry": pattern_registry,
            "point_patterns": point_patterns,
            "empty_policy_result": empty_policy_result,
            "avg_confidence": avg_confidence,
            "deduplicated_tables": dedup_tables,
            "indicator_sample": indicator_dict,
            "proportionality_recommendations": proportionality_recommendations,
            "adjusted_domain_weights": adjusted_weights,
            "posterior_proportionality": posterior_probability,
            "coverage_interval": confidence_interval
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "targets_analyzed": len(context.get("product_targets", [])),
                "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
                "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q3_TraceabilityValidator(BaseExecutor):
    """
    DIM03_Q03_TRACEABILITY_BUDGET_ORG — Validates budgetary and organizational
traceability of products under canonical D3 notation.
    Epistemic mix: structural budget tracing, organizational semantics, and accountability
 synthesis.

    Methods (from D3-Q3):
    - PolicyAnalysisEmbedder.process_document
    - PolicyAnalysisEmbedder.semantic_search
    - PolicyAnalysisEmbedder._apply_mmr
    - PolicyAnalysisEmbedder._generate_query_from_pdq
    - SemanticAnalyzer._empty_semantic_cube
    - FinancialAuditor._match_program_to_node
    - FinancialAuditor._match_goal_to_budget
    - PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
    - PDETMunicipalPlanAnalyzer._consolidate_entities
    - AdaptivePriorCalculator.generate_traceability_record
    - PolicyAnalysisEmbedder.generate_pdq_report
    - ReportingEngine.generate_accountability_matrix
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
```

```
raw_evidence = {}
dim_info = get_dimension_info(CanonicalDimension.D3.value)
document_text = context.get("document_text", "")
document_metadata = context.get("metadata", {})

# Step 1: Match programs to budget nodes
program_matches = self._execute_method(
    "FinancialAuditor", "_match_program_to_node", context
)
goal_budget_matches = self._execute_method(
    "FinancialAuditor", "_match_goal_to_budget", context,
    programs=program_matches
)

# Step 2: Extract responsibility assignments
responsibility_data = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
)
consolidated_entities = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
    entities=responsibility_data
)
responsible_entities = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
)
responsibility_clarity = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
    entities=consolidated_entities
)
# Semantic traceability via embeddings
semantic_chunks = self._execute_method(
    "PolicyAnalysisEmbedder", "process_document", context,
    document_text=document_text,
    document_metadata=document_metadata
)
pdq_query = self._execute_method(
    "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
    pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
)
semantic_hits = self._execute_method(
    "PolicyAnalysisEmbedder", "semantic_search", context,
    query=pdq_query,
    document_chunks=semantic_chunks or []
)
diversified_hits = self._execute_method(
    "PolicyAnalysisEmbedder", "_apply_mmr", context,
    ranked_results=semantic_hits or []
)
semantic_cube_stub = self._execute_method(
    "SemanticAnalyzer", "_empty_semantic_cube", context
)
domain_scores = self._execute_method(
    "SemanticAnalyzer", "_classify_policy_domain", context,
    segment=document_text
)
cross_cutting = self._execute_method(
    "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
    segment=document_text
)
value_chain = self._execute_method(
    "SemanticAnalyzer", "_classify_value_chain_link", context,
    segment=document_text
)
semantic_vectors = self._execute_method(
    "SemanticAnalyzer", "_vectorize_segments", context,
    segments=[document_text]
)
processed_segment = self._execute_method(
```

```
            "SemanticAnalyzer", "_process_segment", context,
            segment=document_text,
            idx=0,
            vector=semantic_vectors[0] if semantic_vectors else None
        )
        semantic_complexity = self._execute_method(
            "SemanticAnalyzer", "_calculate_semantic_complexity", context,
            semantic_cube=semantic_cube_stub
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=[m.get("bpin","") for m in program_matches if isinstance(m, dict)],
            text_length=len(document_text),
            pattern_specificity=0.5
        )
        entity_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_entity_to_dict", context,
entity=e)
            for e in consolidated_entities[:5]
            if isinstance(e, dict) or hasattr(e, "__dict__")
        ]

        # Step 3: Generate traceability records
        traceability_record = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            matches=program_matches
        )

        # Step 4: Generate PDQ report
        pdq_report = self._execute_method(
            "PolicyAnalysisEmbedder", "generate_pdq_report", context,
            traceability=traceability_record
        )

        # Step 5: Generate accountability matrix
        accountability_matrix = self._execute_method(
            "ReportingEngine", "generate_accountability_matrix", context,
            entities=consolidated_entities
        )

        raw_evidence = {
            "budgetary_traceability": {
                "bpin_codes": [m.get("bpin") for m in program_matches if m.get("bpin")],
                "project_codes": [m.get("project_code") for m in program_matches if
m.get("project_code")],
                "budget_matches": goal_budget_matches
            },
            "organizational_traceability": {
                "responsible_entities": consolidated_entities,
                "office_assignments": [e for e in consolidated_entities if
e.get("office")],
                "secretariat_assignments": [e for e in consolidated_entities if
e.get("secretariat")]
            },
            "traceability_record": traceability_record,
            "pdq_report": pdq_report,
            "accountability_matrix": accountability_matrix,
            "responsible_entities": responsible_entities,
            "responsibility_clarity_score": responsibility_clarity,
            "semantic_traceability": {
                "query": pdq_query,
                "semantic_hits": semantic_hits,
                "diversified_hits": diversified_hits
            },
            "semantic_cube_baseline": semantic_cube_stub,
            "policy_domain_scores": domain_scores,
            "responsibility_entities_dict": entity_dicts,
            "cross_cutting_themes": cross_cutting,
```

```
            "value_chain_links": value_chain,
            "semantic_vectors": semantic_vectors,
            "semantic_complexity": semantic_complexity,
            "evidence_confidence": evidence_confidence,
            "processed_segment": processed_segment
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "products_with_bpin": len([m for m in program_matches if m.get("bpin")]),
                "products_with_responsible": len(consolidated_entities),
                "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
    """
    DIM03_Q04_TECHNICAL_FEASIBILITY — Evaluates activity-product feasibility vs
resources/deadlines (canonical D3).
    Epistemic mix: structural DAG validity, causal necessity, performance/implementation
readiness, and statistical robustness.

    Methods (from D3-Q4):
    - AdvancedDAGValidator._calculate_statistical_power
    - AdvancedDAGValidator._initialize_rng
    - AdvancedDAGValidator.export_nodes
    - AdvancedDAGValidator._generate_subgraph
    - AdvancedDAGValidator._get_node_validator
    - AdvancedDAGValidator._create_empty_result
    - HierarchicalGenerativeModel._calculate_likelihood
    - IndustrialGradeValidator.validate_causal_categories
    - TeoriaCambio._extraer_categorias
    - AdvancedDAGValidator.get_graph_stats
    - AdvancedDAGValidator._calculate_node_importance
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - AdvancedDAGValidator._is_acyclic
    - BayesianMechanismInference._test_necessity
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_connection_matrix
    - IndustrialGradeValidator.run_performance_benchmarks
    - IndustrialGradeValidator._benchmark_operation
    - PerformanceAnalyzer.analyze_performance
    - PerformanceAnalyzer._calculate_loss_functions
    - HierarchicalGenerativeModel._calculate_ess
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)
        plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")

        # Step 1: Validate DAG structure
        acyclicity_pvalue = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )
        is_acyclic = self._execute_method(
            "AdvancedDAGValidator", "_is_acyclic", context
        )
```

```
graph_stats = self._execute_method(
    "AdvancedDAGValidator", "get_graph_stats", context
)
node_importance = self._execute_method(
    "AdvancedDAGValidator", "_calculate_node_importance", context
)
subgraph = self._execute_method(
    "AdvancedDAGValidator", "_generate_subgraph", context
)
added_node = self._execute_method(
    "AdvancedDAGValidator", "add_node", context,
    node_name="temp_node"
)
added_edge = self._execute_method(
    "AdvancedDAGValidator", "add_edge", context,
    source="temp_node",
    target="temp_target",
    weight=1.0
)
node_export = self._execute_method(
    "AdvancedDAGValidator", "export_nodes", context
)
rng_seed = self._execute_method(
    "AdvancedDAGValidator", "_initialize_rng", context,
    plan_name=plan_name,
    salt=dim_info.code
)
stat_power = self._execute_method(
    "AdvancedDAGValidator", "_calculate_statistical_power", context,
    s=int(graph_stats.get("edges", 0)),
    n=max(1, int(graph_stats.get("nodes", 1)))
)
node_validator = self._execute_method(
    "AdvancedDAGValidator", "_get_node_validator", context,
    node_type="producto"
)
empty_result = self._execute_method(
    "AdvancedDAGValidator", "_create_empty_result", context,
    plan_name=plan_name,
    seed=rng_seed,
    timestamp=context.get("metadata", {}).get("timestamp", "")
)

# Step 2: Test necessity of activities for products
necessity_test = self._execute_method(
    "BayesianMechanismInference", "_test_necessity", context
)

# Step 3: Execute industrial-grade validation
validation_suite = self._execute_method(
    "IndustrialGradeValidator", "execute_suite", context
)
connection_validation = self._execute_method(
    "IndustrialGradeValidator", "validate_connection_matrix", context
)
performance_benchmarks = self._execute_method(
    "IndustrialGradeValidator", "run_performance_benchmarks", context
)
benchmark_ops = self._execute_method(
    "IndustrialGradeValidator", "_benchmark_operation", context
)
metric_log = self._execute_method(
    "IndustrialGradeValidator", "_log_metric", context,
    name="custom_latency",
    value=graph_stats.get("edges", 0),
    unit="edges",
    threshold=10.0
)
```

```
        engine_readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 4: Analyze performance
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )
        loss_functions = self._execute_method(
            "PerformanceAnalyzer", "_calculate_loss_functions", context
        )
        # Likelihood estimation for resource adequacy
        resource_likelihood = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_likelihood", context,
            mechanism_type="tecnico",
            observations={"coherence": performance_analysis.get("resource_fit",
{}).get("score", 0.0)}
        )

        # Step 5: Calculate effective sample size
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context,
            chains=[]
        )
        causal_categories_valid = self._execute_method(
            "IndustrialGradeValidator", "validate_causal_categories", context
        )
        extracted_categories = self._execute_method(
            "TeoriaCambio", "_extraer_categorias", context,
            text=context.get("document_text", "")
        )

        raw_evidence = {
            "activity_product_mapping": connection_validation,
            "resource_adequacy": performance_analysis.get("resource_fit", {}),
            "timeline_feasibility": performance_analysis.get("timeline_feasibility", {}),
            "technical_validation": {
                "dag_valid": is_acyclic,
                "acyclicity_p": acyclicity_pvalue,
                "necessity_score": necessity_test,
                "graph_stats": graph_stats,
                "node_importance": node_importance,
                "subgraph_sample": subgraph,
                "added_node": added_node,
                "added_edge": added_edge,
                "node_validator": node_validator,
                "empty_result": empty_result,
                "node_export": node_export,
                "rng_seed": rng_seed,
                "statistical_power": stat_power
            },
            "performance_metrics": {
                "benchmarks": performance_benchmarks,
                "loss_functions": loss_functions,
                "ess": ess,
                "r_hat": r_hat,
                "resource_likelihood": resource_likelihood
            },
            "engine_readiness": engine_readiness,
            "feasibility_score": validation_suite.get("overall_score", 0),
            "causal_categories_valid": causal_categories_valid,
            "extracted_categories": extracted_categories,
            "metric_log": metric_log
        }
```

```
        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "dag_is_valid": is_acyclic,
                "feasibility_score": validation_suite.get("overall_score", 0),
                "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):
    """
    DIM03_Q05_OUTPUT_OUTCOME_LINKAGE — Analyzes mechanisms linking outputs to outcomes
with canonical D3 labeling.
    Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect
estimation, and Bayesian mechanism inference.

Methods (from D3-Q5):
    - PDETMunicipalPlanAnalyzer._identify_confounders
    - PDETMunicipalPlanAnalyzer._effect_to_dict
    - PDETMunicipalPlanAnalyzer._scenario_to_dict
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_recommendations
    - PDETMunicipalPlanAnalyzer._identify_causal_nodes
    - BayesianCounterfactualAuditor._evaluate_factual
    - BayesianCounterfactualAuditor._evaluate_counterfactual
    - CausalExtractor._assess_financial_consistency
    - BayesianMechanismInference._infer_activity_sequence
    - BayesianMechanismInference._generate_necessity_remediation
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - IndustrialPolicyProcessor._load_questionnaire
    - PDETMunicipalPlanAnalyzer.analyze_financial_feasibility
    - PDETMunicipalPlanAnalyzer.construct_causal_dag
    - PDETMunicipalPlanAnalyzer.estimate_causal_effects
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - CausalExtractor._build_type_hierarchy
    - CausalExtractor._check_structural_violation
    - CausalExtractor._calculate_type_transition_prior
    - CausalExtractor._calculate_textual_proximity
    - TeoriaCambio._validar_orden_causal
    - PDETMunicipalPlanAnalyzer._refine_edge_probabilities
    - PolicyAnalysisEmbedder.compare_policy_interventions
    - BayesianMechanismInference.infer_mechanisms
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D3.value)

        # Step 0: Build causal backbone and effects
        financial_analysis = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
        )
        causal_dag = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
            financial_analysis=financial_analysis
        )
        causal_effects = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "estimate_causal_effects", context,
            dag=causal_dag,
```

```
            financial_analysis=financial_analysis
        )
        counterfactuals = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context,
            dag=causal_dag,
            causal_effects=causal_effects,
            financial_analysis=financial_analysis
        )
        simulated_intervention = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
            intervention={},
            dag=causal_dag,
            causal_effects=causal_effects,
            label="baseline"
        )
        causal_nodes = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
            text=context.get("document_text", ""),
            tables=context.get("tables", []),
            financial_analysis=financial_analysis
        )
        confounders = {}
        for effect in causal_effects:
            treatment = effect.treatment if hasattr(effect, "treatment") else None
            outcome = effect.outcome if hasattr(effect, "outcome") else None
            if treatment and outcome:
                confounders[(treatment, outcome)] = self._execute_method(
                    "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
                    treatment=treatment,
                    outcome=outcome,
                    dag=causal_dag
                )
        effect_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context,
effect=effect)
            for effect in causal_effects
        ]
        scenario_dicts = [
            self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict",
context, scenario=scenario)
            for scenario in counterfactuals
        ]
        causal_recommendations = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
            analysis_results={"financial_analysis": financial_analysis, "quality_score":
getattr(causal_dag, 'graph', {})}
        )
        financial_consistency = None
        if refined_edges:
            first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
            source = first_edge.get("source") if isinstance(first_edge, dict) else ""
            target = first_edge.get("target") if isinstance(first_edge, dict) else ""
            financial_consistency = self._execute_method(
                "CausalExtractor", "_assess_financial_consistency", context,
                source=source or "",
                target=target or ""
            )
        factual_eval = None
        counterfactual_eval = None
        if causal_effects:
            first_effect = causal_effects[0]
            target = getattr(first_effect, "outcome", None) or ""
            evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
            factual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_factual", context,
                target=target,
                evidence=evidence
            )
```

```
            counterfactual_eval = self._execute_method(
                "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
                target=target,
                intervention={"shift": 0.1}
            )
        matched_node = None
        try:
            matched_node = self._execute_method(
                "PDETMunicipalPlanAnalyzer", "_match_text_to_node", context,
                text=context.get("document_text", "")[:200],
                nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
            )
        except Exception:
            matched_node = None

        # Step 1: Build type hierarchy
        type_hierarchy = self._execute_method(
            "CausalExtractor", "_build_type_hierarchy", context
        )

        # Step 2: Check structural violations
        structural_violations = self._execute_method(
            "CausalExtractor", "_check_structural_violation", context,
            hierarchy=type_hierarchy
        )

        # Step 3: Calculate transition priors and proximity
        transition_priors = self._execute_method(
            "CausalExtractor", "_calculate_type_transition_prior", context,
            hierarchy=type_hierarchy
        )
        textual_proximity = self._execute_method(
            "CausalExtractor", "_calculate_textual_proximity", context
        )

        # Step 4: Validate causal order
        causal_order_validation = self._execute_method(
            "TeoriaCambio", "_validar_orden_causal", context,
            hierarchy=type_hierarchy
        )

        # Step 5: Refine edge probabilities
        refined_edges = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
            priors=transition_priors
        )

        # Step 6: Compare policy interventions
        intervention_comparison = self._execute_method(
            "PolicyAnalysisEmbedder", "compare_policy_interventions", context
        )

        # Step 7: Infer mechanisms
        mechanisms = self._execute_method(
            "BayesianMechanismInference", "infer_mechanisms", context,
            edges=refined_edges
        )
        mechanism_sample = next(iter(mechanisms.values()), {})
        activity_sequence = self._execute_method(
            "BayesianMechanismInference", "_infer_activity_sequence", context,
            observations=mechanism_sample.get("observations", {}),
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico":
1.0})
        )
        quantified_uncertainty = self._execute_method(
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico":
1.0}),
```

```
    sequence_posterior=mechanism_sample.get("activity_sequence", {}),
    coherence_score=mechanism_sample.get("coherence_score", 0.0)
)
mechanism_observations = self._execute_method(
    "BayesianMechanismInference", "_extract_observations", context,
    node={"id": next(iter(mechanisms.keys()), "")},
    text=context.get("document_text", "")
)
necessity_remediation = self._execute_method(
    "BayesianMechanismInference", "_generate_necessity_remediation", context,
    node_id=next(iter(mechanisms.keys()), ""),
    missing_components=structural_violations
)
questionnaire_stub = self._execute_method(
    "IndustrialPolicyProcessor", "_load_questionnaire", context
)
refutation_checks = None
try:
    confounder_keys = list(confounders.keys())
    first_pair = confounder_keys[0] if confounder_keys else ("", "")
    refutation_checks = self._execute_method(
        "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
        dag=getattr(causal_dag, "graph", None),
        target=first_pair[1],
        treatment=first_pair[0],
        confounders=list(confounders.values())[0] if confounders else []
    )
except Exception:
    refutation_checks = None

raw_evidence = {
    "output_outcome_links": refined_edges,
    "mechanism_explanation": mechanisms,
    "type_hierarchy": type_hierarchy,
    "causal_dag": causal_dag,
    "causal_effects": causal_effects,
    "counterfactuals": counterfactuals,
    "simulated_intervention": simulated_intervention,
    "causal_nodes": causal_nodes,
    "financial_analysis": financial_analysis,
    "causal_validity": {
        "structural_violations": structural_violations,
        "order_valid": causal_order_validation
    },
    "transition_probabilities": transition_priors,
    "textual_proximity": textual_proximity,
    "intervention_comparison": intervention_comparison,
    "confounders": confounders,
    "effect_dicts": effect_dicts,
    "scenario_dicts": scenario_dicts,
    "activity_sequence_sample": activity_sequence,
    "uncertainty_quantified": quantified_uncertainty,
    "mechanism_observations": mechanism_observations,
    "refutation_checks": refutation_checks,
    "necessity_remediation": necessity_remediation,
    "questionnaire_stub": questionnaire_stub,
    "causal_recommendations": causal_recommendations,
    "financial_consistency": financial_consistency,
    "factual_eval": factual_eval,
    "counterfactual_eval": counterfactual_eval,
    "matched_node": matched_node
}

return {
    "executor_id": self.executor_id,
    "raw_evidence": raw_evidence,
    "metadata": {
        "methods_executed": [log["method"] for log in self.execution_log],
```

```
            "mechanisms_identified": len(mechanisms),
            "violations_found": len(structural_violations),
            "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
            "dimension_code": dim_info.code,
            "dimension_label": dim_info.label
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


# ============================================================================
# DIMENSION 4: RESULTS & OUTCOMES
# ============================================================================

class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
    """
    DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS — Validates outcome indicators (baseline,
target, horizon) with canonical D4 notation.
    Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical
performance signals, and indicator quality scoring.

    Methods (from D4-Q1):
    - PDETMunicipalPlanAnalyzer._extract_entities_syntax
    - PDETMunicipalPlanAnalyzer._extract_entities_ner
    - CausalExtractor._calculate_language_specificity
    - CausalExtractor._calculate_composite_likelihood
    - CausalExtractor._calculate_semantic_distance
    - TemporalLogicVerifier._classify_temporal_type
    - PDETMunicipalPlanAnalyzer._score_indicators
    - PDETMunicipalPlanAnalyzer._find_outcome_mentions
    - PDETMunicipalPlanAnalyzer._score_temporal_consistency
    - CausalExtractor._extract_goals
    - CausalExtractor._parse_goal_context
    - CausalExtractor._classify_goal_type
    - TemporalLogicVerifier._parse_temporal_marker
    - TemporalLogicVerifier._extract_resources
    - PerformanceAnalyzer.analyze_performance
    - PerformanceAnalyzer._generate_recommendations
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D4.value)

        # Step 1: Find outcome mentions
        outcome_mentions = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_outcome_mentions", context
        )
        entities_syntax = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context,
            text=context.get("document_text", "")
        )
        entities_ner = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context,
            text=context.get("document_text", "")
        )

        # Step 2: Score temporal consistency
        temporal_consistency = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_temporal_consistency", context,
            outcomes=outcome_mentions
        )

        # Step 3: Extract and classify goals
        goals = self._execute_method(
```

```
            "CausalExtractor", "_extract_goals", context
        )
        goal_contexts = self._execute_method(
            "CausalExtractor", "_parse_goal_context", context,
            goals=goals
        )
        goal_types = self._execute_method(
            "CausalExtractor", "_classify_goal_type", context,
            goals=goals
        )
        semantic_distance = 0.0
        if goal_types and outcome_mentions:
            semantic_distance = self._execute_method(
                "CausalExtractor", "_calculate_semantic_distance", context,
                source=str(goal_types[0]),
                target=str(outcome_mentions[0])
            )

        # Step 4: Parse temporal markers
        temporal_markers = self._execute_method(
            "TemporalLogicVerifier", "_parse_temporal_marker", context,
            contexts=goal_contexts
        )
        temporal_type = self._execute_method(
            "TemporalLogicVerifier", "_classify_temporal_type", context,
            marker=temporal_markers[0] if temporal_markers else ""
        )
        resources_mentioned = self._execute_method(
            "TemporalLogicVerifier", "_extract_resources", context,
            text=context.get("document_text", "")
        )
        precedence_check = self._execute_method(
            "TemporalLogicVerifier", "_should_precede", context,
            marker_a=temporal_markers[0] if temporal_markers else "",
            marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
        )

        # Step 5: Analyze performance
        performance_analysis = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context,
            outcomes=outcome_mentions
        )
        indicator_quality = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_indicators", context
        )
        performance_recommendations = self._execute_method(
            "PerformanceAnalyzer", "_generate_recommendations", context,
            performance_analysis=performance_analysis
        )
        # Semantic certainty for goals
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context,
            keyword=goal_contexts[0] if goal_contexts else "",
            policy_area=context.get("policy_area")
        )
        composite_likelihood = self._execute_method(
            "CausalExtractor", "_calculate_composite_likelihood", context,
            evidence={
                "semantic_distance": indicator_quality if isinstance(indicator_quality,
(int, float)) else 0.0,
                "textual_proximity": performance_analysis.get("coherence_score", 0.0) if
isinstance(performance_analysis, dict) else 0.0,
                "language_specificity": language_specificity,
                "temporal_coherence": temporal_consistency if
isinstance(temporal_consistency, (int, float)) else 0.0
            }
        )
```

```
        raw_evidence = {
            "outcome_indicators": outcome_mentions,
            "indicators_with_baseline": [o for o in outcome_mentions if
o.get("has_baseline")],
            "indicators_with_target": [o for o in outcome_mentions if
o.get("has_target")],
            "indicators_with_horizon": [o for o in outcome_mentions if
o.get("time_horizon")],
            "temporal_consistency_score": temporal_consistency,
            "goal_classifications": goal_types,
            "temporal_markers": temporal_markers,
            "performance_metrics": performance_analysis,
            "indicator_quality": indicator_quality,
            "performance_recommendations": performance_recommendations,
            "entities_syntax": entities_syntax,
            "entities_ner": entities_ner,
            "temporal_type": temporal_type,
            "language_specificity": language_specificity,
            "composite_likelihood": composite_likelihood,
            "goal_outcome_semantic_distance": semantic_distance,
            "resources_mentioned": resources_mentioned,
            "precedence_check": precedence_check
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "total_outcomes": len(outcome_mentions),
                "complete_indicators": len([o for o in outcome_mentions
                    if o.get("has_baseline") and o.get("has_target") and
o.get("time_horizon")]),
                "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q2_CausalChainValidator(BaseExecutor):
    """
    Validates explicit causal chain with assumptions and enabling conditions.

    Methods (from D4-Q2):
    - TeoriaCambio._encontrar_caminos_completos
    - TeoriaCambio.validacion_completa
    - CausalExtractor.extract_causal_hierarchy
    - HierarchicalGenerativeModel.verify_conditional_independence
    - HierarchicalGenerativeModel._generate_independence_tests
    - BayesianCounterfactualAuditor.construct_scm
    - AdvancedDAGValidator._perform_sensitivity_analysis_internal
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Find complete causal paths
        complete_paths = self._execute_method(
            "TeoriaCambio", "_encontrar_caminos_completos", context
        )

        # Step 2: Complete validation
```

```
        validation_results = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            paths=complete_paths
        )

        # Step 3: Extract causal hierarchy
        causal_hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        # Step 4: Verify conditional independence
        independence_verification = self._execute_method(
            "HierarchicalGenerativeModel", "verify_conditional_independence", context,
            hierarchy=causal_hierarchy
        )
        independence_tests = self._execute_method(
            "HierarchicalGenerativeModel", "_generate_independence_tests", context,
            verification=independence_verification
        )

        # Step 5: Construct structural causal model
        scm = self._execute_method(
            "BayesianCounterfactualAuditor", "construct_scm", context,
            hierarchy=causal_hierarchy
        )

        # Step 6: Perform sensitivity analysis
        sensitivity_analysis = self._execute_method(
            "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
            scm=scm
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            analysis=sensitivity_analysis
        )

        raw_evidence = {
            "causal_chain": complete_paths,
            "key_assumptions": validation_results.get("assumptions", []),
            "enabling_conditions": validation_results.get("conditions", []),
            "external_factors": validation_results.get("external_factors", []),
            "causal_hierarchy": causal_hierarchy,
            "independence_tests": independence_tests,
            "structural_model": scm,
            "sensitivity": sensitivity_analysis,
            "evidential_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "complete_paths_found": len(complete_paths),
                "assumptions_identified": len(validation_results.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
    """
    Analyzes justification of result ambition based on investment/capacity/benchmarks.
```

```
Methods (from D4-Q3):
- PDETMunicipalPlanAnalyzer._get_prior_effect
- PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
- PDETMunicipalPlanAnalyzer._compute_robustness_value
- AdaptivePriorCalculator.sensitivity_analysis
- HierarchicalGenerativeModel._calculate_r_hat
- HierarchicalGenerativeModel._calculate_ess
- AdvancedDAGValidator._calculate_statistical_power
- BayesianMechanismInference._aggregate_bayesian_confidence
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Get prior effect estimates
    prior_effects = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
    )

    # Step 2: Estimate effect using Bayesian methods
    effect_estimate = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
        priors=prior_effects
    )

    # Step 3: Compute robustness
    robustness = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
        estimate=effect_estimate
    )

    # Step 4: Sensitivity analysis
    sensitivity = self._execute_method(
        "AdaptivePriorCalculator", "sensitivity_analysis", context,
        estimate=effect_estimate
    )

    # Step 5: Calculate convergence diagnostics
    r_hat = self._execute_method(
        "HierarchicalGenerativeModel", "_calculate_r_hat", context
    )
    ess = self._execute_method(
        "HierarchicalGenerativeModel", "_calculate_ess", context
    )

    # Step 6: Calculate statistical power
    statistical_power = self._execute_method(
        "AdvancedDAGValidator", "_calculate_statistical_power", context,
        effect=effect_estimate
    )

    # Step 7: Aggregate confidence
    confidence_aggregate = self._execute_method(
        "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
        estimates=[effect_estimate, robustness, statistical_power]
    )

    raw_evidence = {
        "ambition_level": context.get("target_ambition", {}),
        "financial_investment": context.get("total_investment", 0),
        "institutional_capacity": context.get("capacity_score", 0),
        "comparative_benchmarks": prior_effects,
        "justification_analysis": {
            "effect_estimate": effect_estimate,
            "robustness": robustness,
            "sensitivity": sensitivity,
            "statistical_power": statistical_power
```

```
            },
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess
            },
            "overall_confidence": confidence_aggregate
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "ambition_justified": confidence_aggregate > 0.7,
                "statistical_power": statistical_power
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
    """
    Evaluates whether results address/resolve prioritized problems from diagnosis.

    Methods (from D4-Q4):
    - PolicyContradictionDetector._calculate_objective_alignment
    - PolicyContradictionDetector._identify_affected_sections
    - PolicyContradictionDetector._generate_resolution_recommendations
    - OperationalizationAuditor._generate_optimal_remediations
    - OperationalizationAuditor._get_remediation_text
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - FinancialAuditor._detect_allocation_gaps
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate objective alignment
        objective_alignment = self._execute_method(
            "PolicyContradictionDetector", "_calculate_objective_alignment", context
        )

        # Step 2: Identify affected sections
        affected_sections = self._execute_method(
            "PolicyContradictionDetector", "_identify_affected_sections", context,
            alignment=objective_alignment
        )

        # Step 3: Generate resolution recommendations
        resolutions = self._execute_method(
            "PolicyContradictionDetector", "_generate_resolution_recommendations",
context,
            sections=affected_sections
        )

        # Step 4: Generate optimal remediations
        remediations = self._execute_method(
            "OperationalizationAuditor", "_generate_optimal_remediations", context
        )
        remediation_text = self._execute_method(
            "OperationalizationAuditor", "_get_remediation_text", context,
            remediations=remediations
        )

        # Step 5: Aggregate risk and prioritize
```

```
        risk_priorities = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
        )

        # Step 6: Detect allocation gaps
        allocation_gaps = self._execute_method(
            "FinancialAuditor", "_detect_allocation_gaps", context
        )

        raw_evidence = {
            "prioritized_problems": context.get("diagnosis_problems", []),
            "proposed_results": context.get("outcome_indicators", []),
            "problem_result_mapping": objective_alignment,
            "unaddressed_problems": [p for p in affected_sections if not
p.get("addressed")],
            "solvency_score": objective_alignment.get("score", 0),
            "resolution_recommendations": resolutions,
            "remediations": remediation_text,
            "risk_priorities": risk_priorities,
            "allocation_gaps": allocation_gaps
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "problems_addressed": len([p for p in affected_sections if
p.get("addressed")]),
                "problems_unaddressed": len([p for p in affected_sections if not
p.get("addressed")])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
    """
    Validates alignment with superior frameworks (PND, SDGs).

    Methods (from D4-Q5):
    - PDETMunicipalPlanAnalyzer._score_pdet_alignment
    - PDETMunicipalPlanAnalyzer._score_causal_coherence
    - CDAFFramework._validate_dnp_compliance
    - CDAFFramework._generate_dnp_report
    - IndustrialPolicyProcessor._analyze_causal_dimensions
    - AdaptivePriorCalculator.validate_quality_criteria
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Score PDET alignment
        pdet_alignment = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_pdet_alignment", context
        )

        # Step 2: Score causal coherence
        causal_coherence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
        )

        # Step 3: Validate DNP compliance
        dnp_compliance = self._execute_method(
            "CDAFFramework", "_validate_dnp_compliance", context
```

```
        )
        dnp_report = self._execute_method(
            "CDAFFramework", "_generate_dnp_report", context,
            compliance=dnp_compliance
        )

        # Step 4: Analyze causal dimensions
        causal_dimensions = self._execute_method(
            "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
        )

        # Step 5: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context,
            alignment=pdet_alignment
        )

        raw_evidence = {
            "pnd_alignment": dnp_compliance,
            "sdg_alignment": context.get("sdg_mappings", []),
            "pdet_alignment": pdet_alignment,
            "alignment_declarations": dnp_report.get("declarations", []),
            "causal_coherence": causal_coherence,
            "causal_dimensions": causal_dimensions,
            "quality_validation": quality_validation,
            "alignment_score": (pdet_alignment.get("score", 0) +
                                dnp_compliance.get("score", 0)) / 2
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "pnd_aligned": dnp_compliance.get("is_compliant", False),
                "sdgs_referenced": len(context.get("sdg_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 5: IMPACTS
# ============================================================================

class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
    """
    Analyzes long-term impacts, transmission routes, and time lags.

    Methods (from D5-Q1):
    - PDETMunicipalPlanAnalyzer.generate_counterfactuals
    - PDETMunicipalPlanAnalyzer._simulate_intervention
    - PDETMunicipalPlanAnalyzer._generate_scenario_narrative
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - TeoriaCambio._validar_orden_causal
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine._generate_interventions
    - BayesianCounterfactualAuditor.construct_scm
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Generate counterfactuals
        counterfactuals = self._execute_method(
```

```
    "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context
)

# Step 2: Simulate interventions
simulation = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
    counterfactuals=counterfactuals
)

# Step 3: Generate scenario narratives
scenarios = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_generate_scenario_narrative", context,
    simulation=simulation
)

# Step 4: Find mediator mentions
mediators = self._execute_method(
    "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
)

# Step 5: Validate causal order
causal_order = self._execute_method(
    "TeoriaCambio", "_validar_orden_causal", context,
    mediators=mediators
)

# Step 6: Assess temporal coherence
temporal_coherence = self._execute_method(
    "CausalExtractor", "_assess_temporal_coherence", context
)

# Step 7: Generate interventions
interventions = self._execute_method(
    "TextMiningEngine", "_generate_interventions", context
)

# Step 8: Construct SCM
scm = self._execute_method(
    "BayesianCounterfactualAuditor", "construct_scm", context,
    order=causal_order
)

raw_evidence = {
    "long_term_impacts": context.get("impact_indicators", []),
    "structural_transformations": scenarios,
    "transmission_routes": mediators,
    "expected_time_lags": temporal_coherence.get("time_lags", []),
    "counterfactual_analysis": counterfactuals,
    "simulation_results": simulation,
    "causal_pathways": scm,
    "intervention_scenarios": interventions
}

return {
    "executor_id": self.executor_id,
    "raw_evidence": raw_evidence,
    "metadata": {
        "methods_executed": [log["method"] for log in self.execution_log],
        "impacts_defined": len(context.get("impact_indicators", [])),
        "mediators_identified": len(mediators)
    },
    "execution_metrics": {
        "methods_count": len(self.execution_log),
        "all_succeeded": all(log["success"] for log in self.execution_log)
    }
}
```

```
class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
    """
    DIM05_Q02_COMPOSITE_PROXY_VALIDITY — Validates composite indices/proxies for complex
impacts (canonical D5).
    Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative
reporting quality, and semantic consistency.

    Methods (from D5-Q2):
    - PDETMunicipalPlanAnalyzer._quality_to_dict
    - PolicyAnalysisEmbedder.process_document
    - PolicyAnalysisEmbedder._filter_by_pdq
    - PolicyAnalysisEmbedder._extract_numerical_values
    - PolicyAnalysisEmbedder._compute_overall_confidence
    - PolicyAnalysisEmbedder._embed_texts
    - PolicyTextProcessor.normalize_unicode
    - PolicyTextProcessor.segment_into_sentences
    - PolicyTextProcessor.compile_pattern
    - PolicyTextProcessor.extract_contextual_window
    - IndustrialPolicyProcessor._compute_evidence_confidence
    - IndustrialPolicyProcessor._compute_avg_confidence
    - IndustrialPolicyProcessor._construct_evidence_bundle
    - PDETMunicipalPlanAnalyzer.generate_executive_report
    - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._interpret_overall_quality
    - PolicyAnalysisEmbedder.get_diagnostics
    - PDETMunicipalPlanAnalyzer.calculate_quality_score
    - PDETMunicipalPlanAnalyzer._estimate_score_confidence
    - PDETMunicipalPlanAnalyzer._compute_e_value
    - PDETMunicipalPlanAnalyzer._compute_robustness_value
    - ReportingEngine._calculate_quality_score
    - BayesianMechanismInference._aggregate_bayesian_confidence
    - PolicyAnalysisEmbedder.evaluate_policy_numerical_consistency
    - FinancialAuditor._calculate_sufficiency
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}
        dim_info = get_dimension_info(CanonicalDimension.D5.value)
        document_text = context.get("document_text", "")
        document_metadata = context.get("metadata", {})

        # Step 1: Calculate quality scores
        quality_score = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context
        )
        score_confidence = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_estimate_score_confidence", context,
            score=quality_score
        )

        # Step 2: Compute robustness metrics
        e_value = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
            score=quality_score
        )
        robustness = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
            score=quality_score
        )
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            e_value=e_value,
            robustness=robustness
        )

        # Step 3: Calculate reporting quality score
        reporting_quality = self._execute_method(
```

```
            "ReportingEngine", "_calculate_quality_score", context
        )

        # Step 4: Aggregate Bayesian confidence
        bayesian_confidence = self._execute_method(
            "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
            scores=[quality_score, reporting_quality]
        )

        # Step 5: Evaluate numerical consistency
        numerical_consistency = self._execute_method(
            "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context
        )
        embedder_diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context
        )
        processed_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "process_document", context,
            document_text=document_text,
            document_metadata=document_metadata
        )
        pdq_filter = self._execute_method(
            "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
            pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
        )
        filtered_chunks = self._execute_method(
            "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
            chunks=processed_chunks,
            pdq_filter=pdq_filter
        )
        numerical_values = self._execute_method(
            "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
            chunks=processed_chunks
        )
        embedded_texts = self._execute_method(
            "PolicyAnalysisEmbedder", "_embed_texts", context,
            texts=[c.get("content", "") for c in processed_chunks] if
isinstance(processed_chunks, list) else []
        )
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else
[],
            numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict)
else {"evidence_strength": "weak", "numerical_coherence": 0.0}
        )

        # Step 6: Calculate sufficiency
        sufficiency = self._execute_method(
            "FinancialAuditor", "_calculate_sufficiency", context
        )
        overall_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_overall_quality", context,
            score=getattr(quality_score, "overall_score", quality_score)
        )
        risk_prioritization = self._execute_method(
            "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
            omission_score=1 - quality_score.financial_feasibility if
hasattr(quality_score, "financial_feasibility") else 0.2,
            insufficiency_score=1 - sufficiency.get("coverage_ratio", 0.0),
            unnecessity_score=1 - (robustness if isinstance(robustness, (int, float)) else
 0.0),
            causal_effect=e_value,
            feasibility=quality_score.financial_feasibility if hasattr(quality_score,
"financial_feasibility") else 0.8,
            cost=1.0
        )
        normalized_text = self._execute_method(
```

```
            "PolicyTextProcessor", "normalize_unicode", context,
            text=document_text
        )
        segmented_sentences = self._execute_method(
            "PolicyTextProcessor", "segment_into_sentences", context,
            text=document_text
        )
        evidence_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
            matches=context.get("proxy_indicators", []),
            text_length=len(document_text),
            pattern_specificity=0.5
        )
        avg_confidence = self._execute_method(
            "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
            dimension_analysis={"D5": {"dimension_confidence":
bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence,
dict) else 0.0}}
        )
        quality_dict = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
            quality=quality_score
        )
        evidence_bundle = self._execute_method(
            "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
            dimension=None,
            category="composite",
            matches=context.get("proxy_indicators", []),
            positions=[],
            confidence=bayesian_confidence.get("numerical_coherence", 0.0) if
isinstance(bayesian_confidence, dict) else 0.0
        )
        compiled_pattern = self._execute_method(
            "PolicyTextProcessor", "compile_pattern", context,
            pattern_str=r"[A-Z]{2,}\\s+\\d+"
        )
        contextual_window = self._execute_method(
            "PolicyTextProcessor", "extract_contextual_window", context,
            text=document_text,
            match_position=0,
            window_size=200
        )
        exec_report = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
            analysis_results={"quality_score": quality_dict, "financial_analysis":
context.get("financial_analysis", {}) or {"total_budget": 0, "funding_sources": {},
"confidence": (0, 0)}}
        )
        export_result = self._execute_method(
            "IndustrialPolicyProcessor", "export_results", context,
            results={"quality": quality_dict, "robustness": robustness},
            output_path="output/composite_results.json"
        )

        raw_evidence = {
            "composite_indices": context.get("composite_indicators", []),
            "proxy_indicators": context.get("proxy_indicators", []),
            "validity_justification": score_confidence,
            "robustness_metrics": {
                "e_value": e_value,
                "robustness": robustness,
                "interpretation": sensitivity_interpretation
            },
            "quality_scores": {
                "overall": quality_score,
                "reporting": reporting_quality
            },
            "bayesian_confidence": bayesian_confidence,
```

```
            "numerical_consistency": numerical_consistency,
            "measurement_sufficiency": sufficiency,
            "embedder_diagnostics": embedder_diagnostics,
            "quality_interpretation": overall_interpretation,
            "pdq_filter": pdq_filter,
            "filtered_chunks": filtered_chunks,
            "numerical_values": numerical_values,
            "embedded_texts": embedded_texts,
            "overall_confidence": overall_confidence,
            "risk_prioritization": risk_prioritization,
            "normalized_text": normalized_text,
            "segmented_sentences": segmented_sentences,
            "evidence_confidence": evidence_confidence,
            "avg_confidence": avg_confidence,
            "quality_dict": quality_dict,
            "compiled_pattern": compiled_pattern,
            "contextual_window": contextual_window,
            "evidence_bundle": evidence_bundle,
            "executive_report": exec_report,
            "export_result": export_result
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "composite_indices_count": len(context.get("composite_indicators", [])),
                "validity_score": score_confidence,
                "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
                "dimension_code": dim_info.code,
                "dimension_label": dim_info.label
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
    """
    Analyzes proxy indicators for intangible impacts with validity documentation.

    Methods (from D5-Q3):
    - CausalExtractor._calculate_semantic_distance
    - SemanticAnalyzer.extract_semantic_cube
    - BayesianMechanismInference._quantify_uncertainty
    - PDETMunicipalPlanAnalyzer._find_mediator_mentions
    - PolicyAnalysisEmbedder.get_diagnostics
    - AdaptivePriorCalculator._perturb_evidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate semantic distance
        semantic_distance = self._execute_method(
            "CausalExtractor", "_calculate_semantic_distance", context
        )

        # Step 2: Extract semantic cube
        semantic_cube = self._execute_method(
            "SemanticAnalyzer", "extract_semantic_cube", context
        )

        # Step 3: Quantify uncertainty
        uncertainty = self._execute_method(
```

```
            "BayesianMechanismInference", "_quantify_uncertainty", context,
            semantic_data=semantic_cube
        )

        # Step 4: Find mediator mentions
        mediators = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context
        )

        # Step 5: Get diagnostics
        diagnostics = self._execute_method(
            "PolicyAnalysisEmbedder", "get_diagnostics", context,
            mediators=mediators
        )

        # Step 6: Perturb evidence for sensitivity
        perturbed_evidence = self._execute_method(
            "AdaptivePriorCalculator", "_perturb_evidence", context,
            diagnostics=diagnostics
        )

        raw_evidence = {
            "intangible_impacts": context.get("intangible_indicators", []),
            "proxy_indicators": context.get("proxy_mappings", []),
            "validity_documentation": diagnostics,
            "limitations_acknowledged": diagnostics.get("limitations", []),
            "semantic_relationships": semantic_cube,
            "semantic_distance": semantic_distance,
            "uncertainty_quantification": uncertainty,
            "sensitivity_to_proxies": perturbed_evidence
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "intangibles_count": len(context.get("intangible_indicators", [])),
                "proxies_defined": len(context.get("proxy_mappings", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
    """
    Evaluates systemic risks that could rupture causal mechanisms.

    Methods (from D5-Q4):
    - OperationalizationAuditor._audit_systemic_risk
    - BayesianCounterfactualAuditor.refutation_and_sanity_checks
    - BayesianCounterfactualAuditor._test_effect_stability
    - PDETMunicipalPlanAnalyzer._interpret_risk
    - PDETMunicipalPlanAnalyzer._interpret_sensitivity
    - PDETMunicipalPlanAnalyzer._break_cycles
    - AdaptivePriorCalculator.sensitivity_analysis
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Audit systemic risks
        systemic_risks = self._execute_method(
            "OperationalizationAuditor", "_audit_systemic_risk", context
        )
```

```
        # Step 2: Refutation and sanity checks
        refutation = self._execute_method(
            "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
            risks=systemic_risks
        )

        # Step 3: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context,
            refutation=refutation
        )

        # Step 4: Interpret risks
        risk_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
            risks=systemic_risks
        )

        # Step 5: Interpret sensitivity
        sensitivity_interpretation = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
            stability=effect_stability
        )

        # Step 6: Break cycles if present
        cycle_breaks = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "_break_cycles", context
        )

        # Step 7: Sensitivity analysis
        sensitivity = self._execute_method(
            "AdaptivePriorCalculator", "sensitivity_analysis", context,
            risks=systemic_risks
        )

        raw_evidence = {
            "macroeconomic_risks": [r for r in systemic_risks if r.get("type") ==
"macroeconomic"],
            "environmental_risks": [r for r in systemic_risks if r.get("type") ==
"environmental"],
            "political_risks": [r for r in systemic_risks if r.get("type") ==
"political"],
            "mechanism_rupture_potential": risk_interpretation.get("rupture_probability",
0),
            "effect_stability": effect_stability,
            "refutation_results": refutation,
            "sensitivity_analysis": sensitivity,
            "cycle_vulnerabilities": cycle_breaks
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "systemic_risks_identified": len(systemic_risks),
                "high_risk_count": len([r for r in systemic_risks if r.get("severity") ==
"high"])
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
```

```
"""
Analyzes realism of impact ambition and potential unintended effects.

Methods (from D5-Q5):
- HierarchicalGenerativeModel.posterior_predictive_check
- HierarchicalGenerativeModel._ablation_analysis
- HierarchicalGenerativeModel._calculate_waic_difference
- AdaptivePriorCalculator._add_ood_noise
- AdaptivePriorCalculator.validate_quality_criteria
- PDETMunicipalPlanAnalyzer._compute_e_value
- PDETMunicipalPlanAnalyzer._compute_robustness_value
- BayesianMechanismInference._calculate_coherence_factor
"""

def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
    raw_evidence = {}

    # Step 1: Posterior predictive check
    predictive_check = self._execute_method(
        "HierarchicalGenerativeModel", "posterior_predictive_check", context
    )

    # Step 2: Ablation analysis
    ablation = self._execute_method(
        "HierarchicalGenerativeModel", "_ablation_analysis", context,
        check=predictive_check
    )

    # Step 3: Calculate WAIC difference
    waic_diff = self._execute_method(
        "HierarchicalGenerativeModel", "_calculate_waic_difference", context,
        ablation=ablation
    )

    # Step 4: Add out-of-distribution noise
    ood_analysis = self._execute_method(
        "AdaptivePriorCalculator", "_add_ood_noise", context
    )

    # Step 5: Validate quality criteria
    quality_validation = self._execute_method(
        "AdaptivePriorCalculator", "validate_quality_criteria", context,
        ood=ood_analysis
    )

    # Step 6: Compute robustness metrics
    e_value = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_compute_e_value", context
    )
    robustness = self._execute_method(
        "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context
    )

    # Step 7: Calculate coherence factor
    coherence = self._execute_method(
        "BayesianMechanismInference", "_calculate_coherence_factor", context
    )

    raw_evidence = {
        "impact_ambition_level": context.get("declared_ambition", 0),
        "realism_assessment": predictive_check.get("realism_score", 0),
        "negative_side_effects": ablation.get("negative_effects", []),
        "limit_hypotheses": quality_validation.get("limits", []),
        "robustness_metrics": {
            "e_value": e_value,
            "robustness": robustness,
            "coherence": coherence
        },
```

```
            "predictive_validity": predictive_check,
            "ablation_results": ablation,
            "model_comparison": waic_diff,
            "ood_sensitivity": ood_analysis
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "realism_score": predictive_check.get("realism_score", 0),
                "side_effects_identified": len(ablation.get("negative_effects", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# DIMENSION 6: CAUSALITY & THEORY OF CHANGE
# ============================================================================

class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
    """
    Builds/validates explicit Theory of Change with diagram and assumptions.

    Methods (from D6-Q1):
    - TeoriaCambio.construir_grafo_causal
    - TeoriaCambio.validacion_completa
    - TeoriaCambio.export_nodes
    - ReportingEngine.generate_causal_diagram
    - ReportingEngine.generate_causal_model_json
    - AdvancedDAGValidator.export_nodes
    - PDETMunicipalPlanAnalyzer.export_causal_network
    - CausalExtractor.extract_causal_hierarchy
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Build causal graph
        causal_graph = self._execute_method(
            "TeoriaCambio", "construir_grafo_causal", context
        )

        # Step 2: Complete validation
        validation = self._execute_method(
            "TeoriaCambio", "validacion_completa", context,
            graph=causal_graph
        )

        # Step 3: Export nodes from Theory of Change
        toc_nodes = self._execute_method(
            "TeoriaCambio", "export_nodes", context,
            graph=causal_graph
        )

        # Step 4: Generate causal diagram
        diagram = self._execute_method(
            "ReportingEngine", "generate_causal_diagram", context,
            graph=causal_graph
        )

        # Step 5: Generate causal model JSON
        model_json = self._execute_method(
```

```
            "ReportingEngine", "generate_causal_model_json", context,
            graph=causal_graph
        )

        # Step 6: Export nodes from DAG validator
        dag_nodes = self._execute_method(
            "AdvancedDAGValidator", "export_nodes", context,
            graph=causal_graph
        )

        # Step 7: Export causal network
        network_export = self._execute_method(
            "PDETMunicipalPlanAnalyzer", "export_causal_network", context,
            graph=causal_graph
        )

        # Step 8: Extract causal hierarchy
        hierarchy = self._execute_method(
            "CausalExtractor", "extract_causal_hierarchy", context
        )

        raw_evidence = {
            "toc_exists": len(causal_graph) > 0,
            "toc_diagram": diagram,
            "toc_json": model_json,
            "causal_graph": causal_graph,
            "nodes": toc_nodes,
            "causes_identified": hierarchy.get("causes", []),
            "mediators_identified": hierarchy.get("mediators", []),
            "assumptions": validation.get("assumptions", []),
            "network_structure": network_export,
            "validation_results": validation
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "nodes_count": len(toc_nodes),
                "assumptions_count": len(validation.get("assumptions", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
    """
    Validates logical proportionality: no leaps, intervention matches result scale.

    Methods (from D6-Q2):
    - BeachEvidentialTest.apply_test_logic
    - BayesianMechanismInference._test_necessity
    - BayesianMechanismInference._test_sufficiency
    - BayesianMechanismInference._calculate_coherence_factor
    - BayesianCounterfactualAuditor._test_effect_stability
    - IndustrialGradeValidator.validate_connection_matrix
    - PolicyAnalysisEmbedder._compute_overall_confidence
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Apply evidential tests
        evidential_tests = self._execute_method(
```

```
            "BeachEvidentialTest", "apply_test_logic", context
        )

        # Step 2: Test necessity
        necessity_test = self._execute_method(
            "BayesianMechanismInference", "_test_necessity", context
        )

        # Step 3: Test sufficiency
        sufficiency_test = self._execute_method(
            "BayesianMechanismInference", "_test_sufficiency", context
        )

        # Step 4: Calculate coherence factor
        coherence_factor = self._execute_method(
            "BayesianMechanismInference", "_calculate_coherence_factor", context,
            necessity=necessity_test,
            sufficiency=sufficiency_test
        )

        # Step 5: Test effect stability
        effect_stability = self._execute_method(
            "BayesianCounterfactualAuditor", "_test_effect_stability", context
        )

        # Step 6: Validate connection matrix
        connection_validation = self._execute_method(
            "IndustrialGradeValidator", "validate_connection_matrix", context
        )

        # Step 7: Compute overall confidence
        overall_confidence = self._execute_method(
            "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
            tests=[necessity_test, sufficiency_test, effect_stability]
        )

        raw_evidence = {
            "logical_leaps_detected": evidential_tests.get("leaps", []),
            "intervention_scale": context.get("intervention_magnitude", 0),
            "result_scale": context.get("result_magnitude", 0),
            "proportionality_ratio": context.get("intervention_magnitude", 0) /
max(context.get("result_magnitude", 1), 1),
            "necessity_score": necessity_test,
            "sufficiency_score": sufficiency_test,
            "coherence_factor": coherence_factor,
            "effect_stability": effect_stability,
            "connection_validation": connection_validation,
            "overall_confidence": overall_confidence,
            "implementation_miracles": [leap for leap in evidential_tests.get("leaps", [])

                                        if leap.get("type") == "miracle"]
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "leaps_detected": len(evidential_tests.get("leaps", [])),
                "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] -
1.0) < 0.5
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }
```

```
class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
    """
    Analyzes validation/testing proposals for weak assumptions before scaling.

    Methods (from D6-Q3):
    - IndustrialGradeValidator.execute_suite
    - IndustrialGradeValidator.validate_engine_readiness
    - IndustrialGradeValidator._benchmark_operation
    - AdaptivePriorCalculator.validate_quality_criteria
    - HierarchicalGenerativeModel._calculate_r_hat
    - HierarchicalGenerativeModel._calculate_ess
    - AdvancedDAGValidator.calculate_acyclicity_pvalue
    - PerformanceAnalyzer.analyze_performance
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Execute validation suite
        validation_suite = self._execute_method(
            "IndustrialGradeValidator", "execute_suite", context
        )

        # Step 2: Validate engine readiness
        readiness = self._execute_method(
            "IndustrialGradeValidator", "validate_engine_readiness", context
        )

        # Step 3: Benchmark operations
        benchmarks = self._execute_method(
            "IndustrialGradeValidator", "_benchmark_operation", context
        )

        # Step 4: Validate quality criteria
        quality_validation = self._execute_method(
            "AdaptivePriorCalculator", "validate_quality_criteria", context
        )

        # Step 5: Calculate convergence diagnostics
        r_hat = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_r_hat", context
        )
        ess = self._execute_method(
            "HierarchicalGenerativeModel", "_calculate_ess", context
        )

        # Step 6: Calculate acyclicity p-value
        acyclicity_p = self._execute_method(
            "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
        )

        # Step 7: Analyze performance
        performance = self._execute_method(
            "PerformanceAnalyzer", "analyze_performance", context
        )

        raw_evidence = {
            "inconsistencies_recognized": validation_suite.get("inconsistencies", []),
            "weak_assumptions": quality_validation.get("weak_assumptions", []),
            "pilot_proposals": context.get("pilot_programs", []),
            "testing_proposals": context.get("testing_plans", []),
            "validation_before_scaling": readiness.get("ready_to_scale", False),
            "validation_results": validation_suite,
            "quality_criteria": quality_validation,
            "convergence_diagnostics": {
                "r_hat": r_hat,
                "ess": ess,
```

```
            "acyclicity_p": acyclicity_p
        },
        "performance_analysis": performance,
        "benchmarks": benchmarks
    }

    return {
        "executor_id": self.executor_id,
        "raw_evidence": raw_evidence,
        "metadata": {
            "methods_executed": [log["method"] for log in self.execution_log],
            "inconsistencies_count": len(validation_suite.get("inconsistencies", [])),
            "pilots_proposed": len(context.get("pilot_programs", []))
        },
        "execution_metrics": {
            "methods_count": len(self.execution_log),
            "all_succeeded": all(log["success"] for log in self.execution_log)
        }
    }


class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):
    """
    Analyzes monitoring system with correction mechanisms and learning processes.

    Methods (from D6-Q4):
    - ConfigLoader.update_priors_from_feedback
    - ConfigLoader.check_uncertainty_reduction_criterion
    - ConfigLoader._save_prior_history
    - ConfigLoader._load_uncertainty_history
    - CDAFFramework._extract_feedback_from_audit
    - AdvancedDAGValidator._calculate_node_importance
    - BayesFactorTable.get_bayes_factor
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Update priors from feedback
        prior_updates = self._execute_method(
            "ConfigLoader", "update_priors_from_feedback", context
        )

        # Step 2: Check uncertainty reduction
        uncertainty_reduction = self._execute_method(
            "ConfigLoader", "check_uncertainty_reduction_criterion", context,
            updates=prior_updates
        )

        # Step 3: Save prior history
        history_saved = self._execute_method(
            "ConfigLoader", "_save_prior_history", context,
            updates=prior_updates
        )

        # Step 4: Load uncertainty history
        uncertainty_history = self._execute_method(
            "ConfigLoader", "_load_uncertainty_history", context
        )

        # Step 5: Extract feedback from audit
        feedback_extracted = self._execute_method(
            "CDAFFramework", "_extract_feedback_from_audit", context
        )

        # Step 6: Calculate node importance
        node_importance = self._execute_method(
            "AdvancedDAGValidator", "_calculate_node_importance", context
```

```
        )

        # Step 7: Get Bayes factor
        bayes_factor = self._execute_method(
            "BayesFactorTable", "get_bayes_factor", context,
            updates=prior_updates
        )

        raw_evidence = {
            "monitoring_system_described": len(context.get("monitoring_indicators", [])) >
0,
            "correction_mechanisms": feedback_extracted.get("mechanisms", []),
            "feedback_loops": feedback_extracted.get("loops", []),
            "learning_processes": feedback_extracted.get("learning", []),
            "prior_updates": prior_updates,
            "uncertainty_reduction": uncertainty_reduction,
            "uncertainty_history": uncertainty_history,
            "node_importance": node_importance,
            "learning_strength": bayes_factor
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "feedback_mechanisms": len(feedback_extracted.get("mechanisms", [])),
                "learning_processes": len(feedback_extracted.get("learning", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
    """
    Evaluates contextual adaptation: differential impacts and territorial constraints.

    Methods (from D6-Q5):
    - CausalExtractor._calculate_language_specificity
    - CausalExtractor._assess_temporal_coherence
    - TextMiningEngine.diagnose_critical_links
    - CausalInferenceSetup.identify_failure_points
    - CausalInferenceSetup._get_dynamics_pattern
    - SemanticProcessor.chunk_text
    - SemanticProcessor._detect_pdm_structure
    - SemanticProcessor._detect_table
    - AdaptivePriorCalculator.generate_traceability_record
    """

    def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
        raw_evidence = {}

        # Step 1: Calculate language specificity
        language_specificity = self._execute_method(
            "CausalExtractor", "_calculate_language_specificity", context
        )

        # Step 2: Assess temporal coherence
        temporal_coherence = self._execute_method(
            "CausalExtractor", "_assess_temporal_coherence", context
        )

        # Step 3: Diagnose critical links
        critical_links = self._execute_method(
            "TextMiningEngine", "diagnose_critical_links", context
```

```
        )

        # Step 4: Identify failure points
        failure_points = self._execute_method(
            "CausalInferenceSetup", "identify_failure_points", context
        )

        # Step 5: Get dynamics pattern
        dynamics_pattern = self._execute_method(
            "CausalInferenceSetup", "_get_dynamics_pattern", context
        )

        # Step 6: Process text structure
        text_chunks = self._execute_method(
            "SemanticProcessor", "chunk_text", context
        )
        pdm_structure = self._execute_method(
            "SemanticProcessor", "_detect_pdm_structure", context,
            chunks=text_chunks
        )
        table_detection = self._execute_method(
            "SemanticProcessor", "_detect_table", context,
            chunks=text_chunks
        )

        # Step 7: Generate traceability record
        traceability = self._execute_method(
            "AdaptivePriorCalculator", "generate_traceability_record", context,
            specificity=language_specificity
        )

        raw_evidence = {
            "context_adaptation": language_specificity.get("adaptation_level", 0),
            "differential_impacts_recognized": critical_links.get("differential_groups",
[]),
            "specific_groups_mentioned": critical_links.get("target_groups", []),
            "territorial_constraints": failure_points.get("territorial", []),
            "local_context_integration": pdm_structure.get("local_sections", []),
            "language_specificity": language_specificity,
            "temporal_coherence": temporal_coherence,
            "dynamics_pattern": dynamics_pattern,
            "structure_analysis": pdm_structure,
            "traceability": traceability
        }

        return {
            "executor_id": self.executor_id,
            "raw_evidence": raw_evidence,
            "metadata": {
                "methods_executed": [log["method"] for log in self.execution_log],
                "groups_identified": len(critical_links.get("target_groups", [])),
                "territorial_constraints": len(failure_points.get("territorial", []))
            },
            "execution_metrics": {
                "methods_count": len(self.execution_log),
                "all_succeeded": all(log["success"] for log in self.execution_log)
            }
        }


# ============================================================================
# EXECUTOR REGISTRY
# ============================================================================

EXECUTOR_REGISTRY = {
    "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
    "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
    "D1-Q3": D1_Q3_BudgetAllocationTracer,
```

```
    "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
    "D1-Q5": D1_Q5_ScopeJustificationValidator,

    "D2-Q1": D2_Q1_StructuredPlanningValidator,
    "D2-Q2": D2_Q2_InterventionLogicInferencer,
    "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
    "D2-Q4": D2_Q4_RiskManagementAnalyzer,
    "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,

    "D3-Q1": D3_Q1_IndicatorQualityValidator,
    "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
    "D3-Q3": D3_Q3_TraceabilityValidator,
    "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
    "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,

    "D4-Q1": D4_Q1_OutcomeMetricsValidator,
    "D4-Q2": D4_Q2_CausalChainValidator,
    "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
    "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
    "D4-Q5": D4_Q5_VerticalAlignmentValidator,

    "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
    "D5-Q2": D5_Q2_CompositeMeasurementValidator,
    "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
    "D5-Q4": D5_Q4_SystemicRiskEvaluator,
    "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,

    "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
    "D6-Q2": D6_Q2_LogicalProportionalityValidator,
    "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
    "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
    "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
}


# ============================================================================
# PHASE 2 ORCHESTRATION
# ============================================================================


def _build_method_executor() -> MethodExecutor:
    """Construct a canonical MethodExecutor via the factory wiring."""
    bundle = build_processor()
    method_executor = getattr(bundle, "method_executor", None)
    if not isinstance(method_executor, MethodExecutor):
        raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor
instance.")
    return method_executor


def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
    """Build canonical metadata block using canonical_notation."""
    metadata: Dict[str, Any] = {}
    try:
        dim_key = executor_id.split("-")[0]
        dim_info = get_dimension_info(dim_key)
        metadata["dimension_code"] = dim_info.code
        metadata["dimension_label"] = dim_info.label
    except Exception:
        pass

    if executor_id in CANONICAL_QUESTION_LABELS:
        metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
    return metadata


def run_phase2_executors(context_package: Dict[str, Any],
                         policy_areas: List[str]) -> Dict[str, Any]:
```

```
    """
    Phase 2 Entry Point: Runs all 30 executors for each policy area.

    Args:
        context_package: Canonical package with document data from Phase 1
        policy_areas: List of policy area identifiers to analyze

    Returns:
        Dict mapping policy_area -> executor_id -> raw_evidence
    """
    results = {}
    method_executor = _build_method_executor()

    for policy_area in policy_areas:
        print(f"\n{'='*80}")
        print(f"Processing Policy Area: {policy_area}")
        print(f"{'='*80}\n")

        # Prepare context for this policy area
        area_context = {
            **context_package,
            "policy_area": policy_area
        }

        # Execute all 30 executors
        area_results = {}
        for executor_id, executor_class in EXECUTOR_REGISTRY.items():
            print(f"Running {executor_id}: {executor_class.__name__}...")

            try:
                # Instantiate executor with config
                config = load_executor_config(executor_id)
                executor = executor_class(executor_id, config,
method_executor=method_executor)

                # Execute and collect results
                result = executor.execute(area_context)
                # Append canonical metadata consistently
                result_metadata = result.get("metadata", {})
                result_metadata.update(_canonical_metadata(executor_id))
                result["metadata"] = result_metadata
                area_results[executor_id] = result

                print(f"  ✓ Success: {len(result['metadata']['methods_executed'])} methods
 executed")

            except ExecutorFailure as e:
                print(f"  ✗ FAILED: {str(e)}")
                raise  # Re-raise to stop execution as per requirement

        results[policy_area] = area_results

    return results


def load_executor_config(executor_id: str) -> Dict[str, Any]:
    """
    Load executor configuration from JSON contract.

    Args:
        executor_id: Executor identifier (e.g., "D1-Q1")

    Returns:
        Configuration dictionary from JSON contract
    """
    import json
    from pathlib import Path
```

```
    config_path = Path(f"config/executor_contracts/{executor_id}.json")

    if not config_path.exists():
        raise FileNotFoundError(f"Executor config not found: {config_path}")

    with open(config_path, 'r', encoding='utf-8') as f:
        return json.load(f)


# ============================================================================
# EXAMPLE USAGE
# ============================================================================

if __name__ == "__main__":
    # Example context package from Phase 1
    context_package = {
        "document_path": "data/pdm_municipality_xyz.pdf",
        "document_text": "...",  # Full document text
        "tables": [],  # Extracted tables from Phase 1
        "embeddings": {},  # Precomputed embeddings
        "entities": [],  # Pre-extracted entities
        "metadata": {
            "municipality": "Municipality XYZ",
            "year": 2024,
            "pages": 150
        }
    }

    # Policy areas to analyze
    policy_areas = [
        "PA01",  # Education
        "PA02",  # Health
        "PA03",  # Infrastructure
        # ... up to 10+ policy areas
    ]

    # Run Phase 2
    try:
        results = run_phase2_executors(context_package, policy_areas)
        print("\n" + "="*80)
        print("PHASE 2 COMPLETED SUCCESSFULLY")
        print("="*80)
        print(f"Processed {len(policy_areas)} policy areas")
        print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")
        print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")

    except ExecutorFailure as e:
        print("\n" + "="*80)
        print("PHASE 2 FAILED")
        print("="*80)
        print(f"Error: {str(e)}")
        print("Execution halted as per requirement: any method failure = executor
failure")

===== FILE: src/saaaaaa/core/orchestrator/factory.py =====
"""
Factory module for core module initialization with dependency injection.

This module is responsible for:
1. Reading data from disk (catalogs, schemas, documents, etc.)
2. Constructing InputContracts for core modules
3. Initializing core modules with injected dependencies
4. Managing I/O operations so core modules remain pure

Architectural Pattern:
- Factory reads from disk
- Factory constructs contracts
- Factory injects dependencies into core modules
```

- Core modules remain I/O-free and testable

QUESTIONNAIRE INTEGRITY PROTOCOL:
- Questionnaire loading is now in questionnaire.py module
- All consumers MUST import from questionnaire module
- Use questionnaire.load_questionnaire() which returns CanonicalQuestionnaire

Version: 3.0.0
Status: Refactored to be fully aligned with questionnaire.py
"""

```
import copy
import hashlib
import json
import logging
import threading
from dataclasses import dataclass
from pathlib import Path
from types import MappingProxyType
from typing import Any, Final, Optional

from ..contracts import (
    CDAFFrameworkInputContract,
    ContradictionDetectorInputContract,
    DocumentData,
    EmbeddingPolicyInputContract,
    PDETAnalyzerInputContract,
    PolicyProcessorInputContract,
    SemanticAnalyzerInputContract,
    SemanticChunkingInputContract,
    TeoriaCambioInputContract,
)
from .core import MethodExecutor, Orchestrator
from .executor_config import ExecutorConfig
from .method_registry import MethodRegistry
from .method_source_validator import MethodSourceValidator

logger = logging.getLogger(__name__)

# Canonical repository root - single source of truth for all file paths
_REPO_ROOT = Path(__file__).resolve().parents[4]
_DEFAULT_DATA_DIR = _REPO_ROOT / "data"


# ============================================================================
# CANONICAL QUESTIONNAIRE MANAGEMENT (MOVED FROM questionnaire.py)
# ============================================================================

# RULE 1: ONE PATH - The ONLY valid questionnaire location
_REPO_ROOT = Path(__file__).resolve().parents[4]
QUESTIONNAIRE_PATH: Final[Path] = _REPO_ROOT / "data" / "questionnaire_monolith.json"

# RULE 2: ONE HASH - Expected SHA-256 hash (MUST match or load fails)
EXPECTED_HASH: Final[str] =
"596d940383dd5bd64a5460eadcb65b9b26b2a7929eea838d2169f0f7cee46986"

# RULE 3: ONE STRUCTURE - Expected question counts
EXPECTED_MICRO_QUESTION_COUNT: Final[int] = 300
EXPECTED_MESO_QUESTION_COUNT: Final[int] = 4
EXPECTED_MACRO_QUESTION_COUNT: Final[int] = 1
EXPECTED_TOTAL_QUESTION_COUNT: Final[int] = 305

@dataclass(frozen=True)
class CanonicalQuestionnaire:
    """Immutable, validated, hash-verified questionnaire."""
    data: MappingProxyType[str, Any]
    sha256: str
    micro_questions: tuple[MappingProxyType, ...]
```

```
    meso_questions: tuple[MappingProxyType, ...]
    macro_question: MappingProxyType | None
    micro_question_count: int
    total_question_count: int
    version: str
    schema_version: str

    def __post_init__(self) -> None:
        """Validate all invariants on construction."""
        if self.sha256 != EXPECTED_HASH:
            raise ValueError(f"QUESTIONNAIRE INTEGRITY VIOLATION: Hash mismatch!")
        if self.micro_question_count != EXPECTED_MICRO_QUESTION_COUNT:
            raise ValueError(f"Expected {EXPECTED_MICRO_QUESTION_COUNT} micro questions,
got {self.micro_question_count}")
        if self.total_question_count != EXPECTED_TOTAL_QUESTION_COUNT:
            raise ValueError(f"Expected {EXPECTED_TOTAL_QUESTION_COUNT} total questions,
got {self.total_question_count}")
        logger.info("canonical_questionnaire_validated sha256=%s version=%s",
self.sha256[:16], self.version)

def _validate_questionnaire_structure(data: dict[str, Any]) -> None:
    """Validate questionnaire structure for required fields and types."""
    if not isinstance(data, dict):
        raise ValueError("Questionnaire must be a dictionary")
    required_keys = ["version", "blocks", "schema_version"]
    if missing := [k for k in required_keys if k not in data]:
        raise ValueError(f"Questionnaire missing keys: {missing}")
    blocks = data["blocks"]
    if not isinstance(blocks, dict) or "micro_questions" not in blocks:
        raise ValueError("blocks.micro_questions is required")
    # (A full validation would check all fields and types recursively)

def _compute_hash(data: dict[str, Any]) -> str:
    """Compute deterministic SHA-256 hash of questionnaire data."""
    canonical_json = json.dumps(data, sort_keys=True, ensure_ascii=True, separators=(',',
':'))
    return hashlib.sha256(canonical_json.encode('utf-8')).hexdigest()

_questionnaire_cache: Optional[CanonicalQuestionnaire] = None

def load_questionnaire() -> CanonicalQuestionnaire:
    """Loads, validates, and caches the questionnaire from the canonical path."""
    global _questionnaire_cache
    if _questionnaire_cache is not None:
        logger.debug("Returning cached canonical questionnaire.")
        return _questionnaire_cache

    path = QUESTIONNAIRE_PATH
    if not path.exists():
        raise FileNotFoundError(f"Canonical questionnaire not found: {path}")

    logger.info(f"Loading canonical questionnaire from {path}")
    content = path.read_text(encoding='utf-8')
    data = json.loads(content)

    _validate_questionnaire_structure(data)
    sha256 = _compute_hash(data)

    blocks = data['blocks']
    micro_questions = tuple(MappingProxyType(q) for q in blocks['micro_questions'])
    meso_questions = tuple(MappingProxyType(q) for q in blocks.get('meso_questions', []))
    macro_question = MappingProxyType(blocks['macro_question']) if 'macro_question' in
blocks else None
    total_count = len(micro_questions) + len(meso_questions) + (1 if macro_question else
0)

    canonical_q = CanonicalQuestionnaire(
        data=MappingProxyType(data),
```

```
        sha256=sha256,
        micro_questions=micro_questions,
        meso_questions=meso_questions,
        macro_question=macro_question,
        micro_question_count=len(micro_questions),
        total_question_count=total_count,
        version=data.get('version', 'unknown'),
        schema_version=data.get('schema_version', 'unknown'),
    )

    _questionnaire_cache = canonical_q
    return canonical_q

# ============================================================================
# END OF MOVED QUESTIONNAIRE LOGIC
# ============================================================================


@dataclass(frozen=True)
class ProcessorBundle:
    """Aggregated orchestrator dependencies built by the factory.

    Attributes:
        method_executor: Preconfigured :class:`MethodExecutor` instance ready for
            execution.
        questionnaire: The canonical, immutable questionnaire object.
        factory: The :class:`CoreModuleFactory` used to construct ancillary
            input contracts for downstream processors.
        signal_registry: Optional signal registry populated during factory wiring.
        executor_config: Canonical :class:`ExecutorConfig` used for all question
            executors.
    """

    method_executor: MethodExecutor
    questionnaire: CanonicalQuestionnaire
    factory: "CoreModuleFactory"
    signal_registry: Any | None
    executor_config: ExecutorConfig

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

def load_catalog(path: Path | None = None) -> dict[str, Any]:
    """Load method catalog JSON file.

    Args:
        path: Path to catalog file. Defaults to
config/rules/METODOS/catalogo_completo_canonico.json
            relative to repository root.

    Returns:
        Loaded catalog data

    Raises:
        FileNotFoundError: If catalog file doesn't exist
        json.JSONDecodeError: If file is not valid JSON
    """
    if path is None:
        path = _REPO_ROOT / "config" / "rules" / "METODOS" /
"catalogo_completo_canonico.json"

    logger.info(f"Loading catalog from {path}")

    with open(path, encoding='utf-8') as f:
        return json.load(f)

def load_method_map(path: Path | None = None) -> dict[str, Any]:
```

```
    """Load method-class mapping JSON file.

    Args:
        path: Path to method map file. Defaults to COMPLETE_METHOD_CLASS_MAP.json
              relative to repository root.

    Returns:
        Loaded method map data

    Raises:
        FileNotFoundError: If method map file doesn't exist
        json.JSONDecodeError: If file is not valid JSON
    """
    if path is None:
        path = _REPO_ROOT / "COMPLETE_METHOD_CLASS_MAP.json"

    logger.info(f"Loading method map from {path}")

    with open(path, encoding='utf-8') as f:
        return json.load(f)

def get_canonical_dimensions(questionnaire_path: Path | None = None) -> dict[str,
dict[str, str]]:
    """
    Get canonical dimension definitions from questionnaire monolith.

    Args:
        questionnaire_path: Optional path to questionnaire file (IGNORED for integrity)

    Returns:
        Dictionary mapping dimension keys (D1-D6) to dimension info.
    """
    if questionnaire_path is not None:
        logger.warning(
            "get_canonical_dimensions: questionnaire_path parameter is IGNORED. "
            "Dimensions always load from canonical questionnaire path for integrity."
        )

    canonical = load_questionnaire()

    if 'canonical_notation' not in canonical.data:
        raise KeyError("canonical_notation section missing from questionnaire")

    if 'dimensions' not in canonical.data['canonical_notation']:
        raise KeyError("dimensions section missing from canonical_notation")

    return copy.deepcopy(canonical.data['canonical_notation']['dimensions'])

def get_canonical_policy_areas(questionnaire_path: Path | None = None) -> dict[str,
dict[str, str]]:
    """
    Get canonical policy area definitions from questionnaire monolith.

    Args:
        questionnaire_path: Optional path to questionnaire file (IGNORED for integrity)

    Returns:
        Dictionary mapping policy area codes (PA01-PA10) to policy area info.
    """
    if questionnaire_path is not None:
        logger.warning(
            "get_canonical_policy_areas: questionnaire_path parameter is IGNORED. "
            "Policy areas always load from canonical questionnaire path for integrity."
        )

    canonical = load_questionnaire()

    if 'canonical_notation' not in canonical.data:
```

```
        raise KeyError("canonical_notation section missing from questionnaire")

    if 'policy_areas' not in canonical.data['canonical_notation']:
        raise KeyError("policy_areas section missing from canonical_notation")

    return copy.deepcopy(canonical.data['canonical_notation']['policy_areas'])

def load_schema(path: Path | None = None) -> dict[str, Any]:
    """Load questionnaire schema JSON file.

    Args:
        path: Path to schema file. Defaults to schemas/questionnaire_monolith.schema.json
              relative to repository root.
    """
    if path is None:
        path = _REPO_ROOT / "schemas" / "questionnaire_monolith.schema.json"

    logger.info(f"Loading schema from {path}")

    with open(path, encoding='utf-8') as f:
        return json.load(f)

def load_document(file_path: Path) -> DocumentData:
    """Load a document and construct DocumentData contract."""
    logger.info(f"Loading document from {file_path}")

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

    sentences = [s.strip() for s in raw_text.split('.') if s.strip()]

    return DocumentData(
        raw_text=raw_text,
        sentences=sentences,
        tables=[],
        metadata={
            'file_path': str(file_path),
            'file_name': file_path.name,
            'num_sentences': len(sentences),
        }
    )

def save_results(results: dict[str, Any], output_path: Path) -> None:
    """Save analysis results to file."""
    logger.info(f"Saving results to {output_path}")

    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(results, f, indent=2, ensure_ascii=False)

# ============================================================================
# CONTRACT CONSTRUCTORS
# ============================================================================


def construct_semantic_analyzer_input(document: DocumentData, **kwargs) ->
SemanticAnalyzerInputContract:
    """Constructs the input for the SemanticAnalyzer."""
    return SemanticAnalyzerInputContract(
        text=document['raw_text'],
        segments=document['sentences'],
        ontology_params=kwargs.get('ontology_params', {})
    )

def construct_cdaf_input(document: DocumentData, **kwargs) -> CDAFFrameworkInputContract:
    """Constructs the input for the CDAFFramework."""
    return CDAFFrameworkInputContract(
        document_text=document['raw_text'],
        plan_metadata=document['metadata'],
```

```
        config=kwargs.get('config', {})
    )

def construct_pdet_input(document: DocumentData, **kwargs) -> PDETAnalyzerInputContract:
    """Constructs the input for the PDETAnalyzer."""
    return PDETAnalyzerInputContract(
        document_content=document['raw_text'],
        extract_tables=kwargs.get('extract_tables', True),
        config=kwargs.get('config', {})
    )

def construct_teoria_cambio_input(document: DocumentData, **kwargs) ->
TeoriaCambioInputContract:
    """Constructs the input for the TeoriaCambio."""
    return TeoriaCambioInputContract(
        document_text=document['raw_text'],
        strategic_goals=kwargs.get('strategic_goals', []),
        config=kwargs.get('config', {})
    )

def construct_contradiction_detector_input(document: DocumentData, **kwargs) ->
ContradictionDetectorInputContract:
    """Constructs the input for the ContradictionDetector."""
    return ContradictionDetectorInputContract(
        text=document['raw_text'],
        plan_name=document['metadata'].get('file_name', 'Unknown'),
        dimension=kwargs.get('dimension'),
        config=kwargs.get('config', {})
    )

def construct_embedding_policy_input(document: DocumentData, **kwargs) ->
EmbeddingPolicyInputContract:
    """Constructs the input for the EmbeddingPolicy."""
    return EmbeddingPolicyInputContract(
        text=document['raw_text'],
        dimensions=kwargs.get('dimensions', []),
        model_config=kwargs.get('model_config', {})
    )

def construct_semantic_chunking_input(document: DocumentData, **kwargs) ->
SemanticChunkingInputContract:
    """Constructs the input for the SemanticChunking."""
    return SemanticChunkingInputContract(
        text=document['raw_text'],
        preserve_structure=kwargs.get('preserve_structure', True),
        config=kwargs.get('config', {})
    )

def construct_policy_processor_input(document: DocumentData, **kwargs) ->
PolicyProcessorInputContract:
    """Constructs the input for the PolicyProcessor."""
    return PolicyProcessorInputContract(
        data={},
        text=document['raw_text'],
        sentences=document['sentences'],
        tables=document['tables'],
        config=kwargs.get('config', {})
    )

# ============================================================================
# FACTORY FUNCTIONS
# ============================================================================

class CoreModuleFactory:
    """Factory for constructing core modules with injected dependencies."""

    def __init__(self, data_dir: Path | None = None) -> None:
        """Initialize factory."""
```

```
        self.data_dir = data_dir or _DEFAULT_DATA_DIR
        self.questionnaire_cache: CanonicalQuestionnaire | None = None
        self.catalog_cache: dict[str, Any] | None = None
        self._lock = threading.Lock()

    def get_questionnaire(self) -> CanonicalQuestionnaire:
        """Get the canonical questionnaire object (cached)."""
        with self._lock:
            if self.questionnaire_cache is None:
                canonical_q = load_questionnaire()
                self.questionnaire_cache = canonical_q
                logger.info(
                    "factory_loaded_questionnaire sha256=%s... question_count=%s",
                    canonical_q.sha256[:16],
                    canonical_q.total_question_count,
                )
            return self.questionnaire_cache

    @property
    def catalog(self) -> dict[str, Any]:
        """Get method catalog data (cached)."""
        with self._lock:
            if self.catalog_cache is None:
                self.catalog_cache = load_catalog()
            return self.catalog_cache

    def load_document(self, file_path: Path) -> DocumentData:
        """Load document and return structured data."""
        return load_document(file_path)

    def save_results(self, results: dict[str, Any], output_path: Path) -> None:
        """Save analysis results."""
        save_results(results, output_path)

    def load_catalog(self, path: Path | None = None) -> dict[str, Any]:
        """Load method catalog JSON file."""
        return load_catalog(path)

    # Contract constructor methods
    construct_semantic_analyzer_input = construct_semantic_analyzer_input
    construct_cdaf_input = construct_cdaf_input
    construct_pdet_input = construct_pdet_input
    construct_teoria_cambio_input = construct_teoria_cambio_input
    construct_contradiction_detector_input = construct_contradiction_detector_input
    construct_embedding_policy_input = construct_embedding_policy_input
    construct_semantic_chunking_input = construct_semantic_chunking_input
    construct_policy_processor_input = construct_policy_processor_input

def build_processor(
    *,
    questionnaire_path: Path | None = None,
    data_dir: Path | None = None,
    factory: Optional["CoreModuleFactory"] = None,
    enable_signals: bool = True,
    executor_config: ExecutorConfig | None = None,
) -> ProcessorBundle:
    """Create a processor bundle with orchestrator dependencies wired together."""

    # PHASE 1: SOURCE-TRUTH VALIDATION
    logger.info("Running source-truth validation...")
    validator = MethodSourceValidator()
    source_truth = validator.generate_source_truth_map()

    # Note: As per user instruction, executors_methods.json is outdated.
    # The validation should eventually be against the docstrings of the executors.
    # For now, we proceed with the validation against the file to identify discrepancies.
    validation_report = validator.validate_executor_methods()
```

```
    if validation_report['missing']:
        # In a strict environment, this should raise an error.
        # For now, we will log a warning to avoid blocking the pipeline.
        logger.warning(f"MISSING METHODS DETECTED: {validation_report['missing']}")
        # raise RuntimeError(f"MISSING METHODS: {validation_report['missing']}")

    # PHASE 2: METHOD REGISTRY CREATION
    logger.info("Initializing method registry with source-truth...")
    method_registry = MethodRegistry()
    logger.info(f"Pre-registered {len(validation_report['valid'])} valid methods.")


    # Runtime type checks
    if questionnaire_path is not None and not isinstance(questionnaire_path, Path):
        raise TypeError(f"questionnaire_path must be Path or None, got
{type(questionnaire_path).__name__}.")
    if data_dir is not None and not isinstance(data_dir, Path):
        raise TypeError(f"data_dir must be Path or None, got {type(data_dir).__name__}")
    if factory is not None and not isinstance(factory, CoreModuleFactory):
        raise TypeError(f"factory must be CoreModuleFactory or None, got
{type(factory).__name__}")
    if not isinstance(enable_signals, bool):
        raise TypeError(f"enable_signals must be bool, got
{type(enable_signals).__name__}")
    if executor_config is not None and not isinstance(executor_config, ExecutorConfig):
        raise TypeError(f"executor_config must be ExecutorConfig or None, got
{type(executor_config).__name__}")

    core_factory = factory or CoreModuleFactory(data_dir=data_dir)
    effective_config = executor_config or ExecutorConfig()

    if questionnaire_path:
        canonical_q = load_questionnaire(questionnaire_path)
        core_factory.questionnaire_cache = canonical_q
        logger.info(
            "build_processor_using_canonical_loader path=%s sha256=%s...
question_count=%s",
            str(questionnaire_path),
            canonical_q.sha256[:16],
            canonical_q.total_question_count,
        )
    else:
        canonical_q = core_factory.get_questionnaire()

    # Build signal infrastructure if enabled
    signal_registry = None
    #if enable_signals:
    #    try:
    #        from .bayesian_module_factory import BayesianModuleFactory as SignalFactory
    #
    #        signal_factory = SignalFactory(
    #            questionnaire_data=canonical_q.data,  # Pass the immutable data view
    #            enable_signals=True,
    #        )
    #        signal_registry = signal_factory._signal_registry
    #
    #        logger.info(
    #            "signals_enabled_in_processor enabled=%s registry_size=%s",
    #            True,
    #            len(signal_registry._cache) if signal_registry else 0,
    #        )
    #    except Exception as e:
    #        logger.warning(
    #            "signal_initialization_failed error=%s fallback=%s",
    #            str(e),
    #            "continuing without signals",
    #        )
    #        signal_registry = None
```

```
    executor = MethodExecutor(
        signal_registry=signal_registry,
        method_registry=method_registry
    )

    return ProcessorBundle(
        method_executor=executor,
        questionnaire=canonical_q,
        factory=core_factory,
        signal_registry=signal_registry,
        executor_config=effective_config,
    )

def create_orchestrator() -> "Orchestrator":
    """Create a fully configured orchestrator instance."""
    processor_bundle = build_processor()
    return Orchestrator(
        method_executor=processor_bundle.method_executor,
        questionnaire=processor_bundle.questionnaire,
        executor_config=processor_bundle.executor_config,
    )

def get_questionnaire_provider():
    """Get a questionnaire provider instance."""
    from ..wiring.bootstrap import QuestionnaireResourceProvider
    return QuestionnaireResourceProvider()

# ============================================================================
# MIGRATION HELPERS
# ============================================================================

def migrate_io_from_module(module_name: str, line_numbers: list[int]) -> None:
    """Helper to track I/O migration progress."""
    logger.info(
        f"Migrating {len(line_numbers)} I/O operations from {module_name}: "
        f"lines {line_numbers}"
    )

__all__ = [
    # Questionnaire integrity types and constants
    'CanonicalQuestionnaire',
    'EXPECTED_HASH',
    'EXPECTED_MACRO_QUESTION_COUNT',
    'EXPECTED_MICRO_QUESTION_COUNT',
    'EXPECTED_MESO_QUESTION_COUNT',
    'EXPECTED_TOTAL_QUESTION_COUNT',
    'QUESTIONNAIRE_PATH',
    'load_questionnaire',
    # Factory classes
    'CoreModuleFactory',
    'ProcessorBundle',
    # Other loaders
    'load_catalog',
    'load_method_map',
    'get_canonical_dimensions',
    'get_canonical_policy_areas',
    'load_schema',
    'load_document',
    'save_results',
    # Contract constructors
    'construct_semantic_analyzer_input',
    'construct_cdaf_input',
    'construct_pdet_input',
    'construct_teoria_cambio_input',
    'construct_contradiction_detector_input',
    'construct_embedding_policy_input',
    'construct_semantic_chunking_input',
```

```
    'construct_policy_processor_input',
    # Builder
    'build_processor',
    'get_questionnaire_provider',
]

===== FILE: src/saaaaaa/core/orchestrator/method_registry.py =====
"""Method Registry with lazy instantiation and injection pattern.

This module implements a method injection factory that:
1. Loads only the methods needed (not full classes)
2. Instantiates classes lazily (only when first method is called)
3. Caches instances for reuse
4. Isolates errors per method (failures don't cascade)
5. Allows direct function injection (bypassing classes)

Architecture:
    MethodRegistry
        ├─ _class_paths: mapping of class names to import paths
        ├─ _instance_cache: lazily instantiated class instances
        ├─ _direct_methods: directly injected functions
        └─ get_method(): returns callable for (class_name, method_name)

Benefits:
- No upfront class loading (lightweight imports)
- Failed classes don't block working methods
- Direct function injection for custom implementations
- Instance reuse through caching
"""
from __future__ import annotations

import logging
import threading
from importlib import import_module
from typing import Any, Callable

logger = logging.getLogger(__name__)


class MethodRegistryError(RuntimeError):
    """Raised when a method cannot be retrieved."""


class MethodRegistry:
    """Registry for lazy method injection without full class instantiation."""

    def __init__(self, class_paths: dict[str, str] | None = None) -> None:
        """Initialize the method registry.

        Args:
            class_paths: Optional mapping of class names to import paths.
                         If None, uses default paths from class_registry.
        """
        # Import class paths from existing registry
        if class_paths is None:
            from .class_registry import get_class_paths
            class_paths = dict(get_class_paths())

        self._class_paths = class_paths
        self._instance_cache: dict[str, Any] = {}
        self._direct_methods: dict[tuple[str, str], Callable[..., Any]] = {}
        self._failed_classes: set[str] = set()
        self._lock = threading.Lock()

        # Special instantiation rules (from original MethodExecutor)
        self._special_instantiation: dict[str, Callable[[type], Any]] = {}

    def inject_method(
```

```
    self,
    class_name: str,
    method_name: str,
    method: Callable[..., Any],
) -> None:
    """Directly inject a method without needing a class.

    This allows bypassing class instantiation entirely.

    Args:
        class_name: Virtual class name for routing
        method_name: Method name
        method: Callable to inject
    """
    key = (class_name, method_name)
    self._direct_methods[key] = method
    logger.info(
        "method_injected_directly",
        class_name=class_name,
        method_name=method_name,
    )

def register_instantiation_rule(
    self,
    class_name: str,
    instantiator: Callable[[type], Any],
) -> None:
    """Register special instantiation logic for a class.

    Args:
        class_name: Class name requiring special instantiation
        instantiator: Function that takes class type and returns instance
    """
    self._special_instantiation[class_name] = instantiator
    logger.debug(
        "instantiation_rule_registered",
        class_name=class_name,
    )

def _load_class(self, class_name: str) -> type:
    """Load a class type from import path.

    Args:
        class_name: Name of class to load

    Returns:
        Class type

    Raises:
        MethodRegistryError: If class cannot be loaded
    """
    if class_name not in self._class_paths:
        raise MethodRegistryError(
            f"Class '{class_name}' not found in registry paths"
        )

    path = self._class_paths[class_name]
    module_name, _, attr_name = path.rpartition(".")

    if not module_name:
        raise MethodRegistryError(
            f"Invalid path for '{class_name}': {path}"
        )

    try:
        module = import_module(module_name)
        cls = getattr(module, attr_name)
```

```
            if not isinstance(cls, type):
                raise MethodRegistryError(
                    f"'{class_name}' is not a class: {type(cls).__name__}"
                )

            return cls

        except ImportError as exc:
            raise MethodRegistryError(
                f"Cannot import class '{class_name}' from {path}: {exc}"
            ) from exc
        except AttributeError as exc:
            raise MethodRegistryError(
                f"Class '{attr_name}' not found in module {module_name}: {exc}"
            ) from exc

    def _instantiate_class(self, class_name: str, cls: type) -> Any:
        """Instantiate a class using special rules or default constructor.

        Args:
            class_name: Name of class (for special rule lookup)
            cls: Class type to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If instantiation fails
        """
        # Use special instantiation rule if registered
        if class_name in self._special_instantiation:
            try:
                instantiator = self._special_instantiation[class_name]
                instance = instantiator(cls)
                logger.debug(
                    "class_instantiated_with_special_rule",
                    class_name=class_name,
                )
                return instance
            except Exception as exc:
                raise MethodRegistryError(
                    f"Special instantiation failed for '{class_name}': {exc}"
                ) from exc

        # Default instantiation (no-args constructor)
        try:
            instance = cls()
            logger.debug(
                "class_instantiated_default",
                class_name=class_name,
            )
            return instance
        except Exception as exc:
            raise MethodRegistryError(
                f"Default instantiation failed for '{class_name}': {exc}"
            ) from exc

    def _get_instance(self, class_name: str) -> Any:
        """Get or create instance of a class (lazy + cached).

        Args:
            class_name: Name of class to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If class cannot be instantiated
```

```
    """
    # Check if already failed
    if class_name in self._failed_classes:
        raise MethodRegistryError(
            f"Class '{class_name}' previously failed to instantiate"
        )

    # Use a lock to ensure thread-safe instantiation
    with self._lock:
        # Double-check if another thread instantiated it while waiting for the lock
        if class_name in self._instance_cache:
            return self._instance_cache[class_name]

        # Load and instantiate class
        try:
            cls = self._load_class(class_name)
            instance = self._instantiate_class(class_name, cls)
            self._instance_cache[class_name] = instance
            logger.info(
                "class_instantiated_lazy",
                class_name=class_name,
            )
            return instance

        except MethodRegistryError:
            # Mark as failed to avoid repeated attempts
            self._failed_classes.add(class_name)
            raise

def get_method(
    self,
    class_name: str,
    method_name: str,
) -> Callable[..., Any]:
    """Get method callable with lazy instantiation.

    This is the main entry point for retrieving methods.

    Args:
        class_name: Name of class containing the method
        method_name: Name of method to retrieve

    Returns:
        Callable method (bound or injected)

    Raises:
        MethodRegistryError: If method cannot be retrieved
    """
    # Check for directly injected method first
    key = (class_name, method_name)
    if key in self._direct_methods:
        logger.debug(
            "method_retrieved_direct",
            class_name=class_name,
            method_name=method_name,
        )
        return self._direct_methods[key]

    # Get instance (lazy) and retrieve method
    try:
        instance = self._get_instance(class_name)
        method = getattr(instance, method_name)

        if not callable(method):
            raise MethodRegistryError(
                f"'{class_name}.{method_name}' is not callable"
            )
```

```
            logger.debug(
                "method_retrieved_from_instance",
                class_name=class_name,
                method_name=method_name,
            )
            return method

        except AttributeError as exc:
            raise MethodRegistryError(
                f"Method '{method_name}' not found on class '{class_name}'"
            ) from exc

    def has_method(self, class_name: str, method_name: str) -> bool:
        """Check if a method is available (without instantiating).

        Args:
            class_name: Name of class
            method_name: Name of method

        Returns:
            True if method exists (or is directly injected)
        """
        # Check direct injection
        key = (class_name, method_name)
        if key in self._direct_methods:
            return True

        # Check if class is known and not failed
        if class_name in self._failed_classes:
            return False

        if class_name not in self._class_paths:
            return False

        # If instance exists, check method
        if class_name in self._instance_cache:
            instance = self._instance_cache[class_name]
            return hasattr(instance, method_name)

        # Otherwise, assume it exists (lazy check)
        # Full validation happens on first get_method() call
        return True

    def get_stats(self) -> dict[str, Any]:
        """Get registry statistics.

        Returns:
            Dictionary with registry stats
        """
        return {
            "total_classes_registered": len(self._class_paths),
            "instantiated_classes": len(self._instance_cache),
            "failed_classes": len(self._failed_classes),
            "direct_methods_injected": len(self._direct_methods),
            "instantiated_class_names": list(self._instance_cache.keys()),
            "failed_class_names": list(self._failed_classes),
        }


def setup_default_instantiation_rules(registry: MethodRegistry) -> None:
    """Setup default special instantiation rules.

    These rules replicate the logic from the original MethodExecutor
    for classes that need non-default instantiation.

    Args:
        registry: MethodRegistry to configure
    """
```

```
    # MunicipalOntology - shared instance pattern
    ontology_instance = None

    def instantiate_ontology(cls: type) -> Any:
        nonlocal ontology_instance
        if ontology_instance is None:
            ontology_instance = cls()
        return ontology_instance

    registry.register_instantiation_rule("MunicipalOntology", instantiate_ontology)

    # SemanticAnalyzer, PerformanceAnalyzer, TextMiningEngine - need ontology
    def instantiate_with_ontology(cls: type) -> Any:
        if ontology_instance is None:
            raise MethodRegistryError(
                f"Cannot instantiate {cls.__name__}: MunicipalOntology not available"
            )
        return cls(ontology_instance)

    for class_name in ["SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine"]:
        registry.register_instantiation_rule(class_name, instantiate_with_ontology)

    # PolicyTextProcessor - needs ProcessorConfig
    def instantiate_policy_processor(cls: type) -> Any:
        try:
            from policy_processor import ProcessorConfig
            return cls(ProcessorConfig())
        except ImportError as exc:
            raise MethodRegistryError(
                "Cannot instantiate PolicyTextProcessor: ProcessorConfig unavailable"
            ) from exc

    registry.register_instantiation_rule("PolicyTextProcessor",
instantiate_policy_processor)


__all__ = [
    "MethodRegistry",
    "MethodRegistryError",
    "setup_default_instantiation_rules",
]

===== FILE: src/saaaaaa/core/orchestrator/method_source_validator.py =====

import ast
import os
import json
from typing import Dict, List, Any

class MethodSourceValidator:
    def __init__(self, base_path: str = "src/saaaaaa"):
        self.base_path = base_path
        self.source_map = self._build_source_map()

    def _build_source_map(self) -> Dict[str, Dict[str, Any]]:
        class_map = {}
        for root, _, files in os.walk(self.base_path):
            for file in files:
                if file.endswith(".py"):
                    file_path = os.path.join(root, file)
                    with open(file_path, "r", encoding="utf-8") as f:
                        try:
                            tree = ast.parse(f.read(), filename=file_path)
                            for node in ast.walk(tree):
                                if isinstance(node, ast.ClassDef):
                                    class_name = node.name
                                    methods = []
                                    for item in node.body:
```

```
                            if isinstance(item, ast.FunctionDef):
                                methods.append(item.name)

                        if class_name in class_map:
                            # In case of duplicate class names, we might need
a more robust way
                            # to handle this, but for now we'll just
overwrite.
                            # A better approach could be to store a list of
locations.
                            pass

                        class_map[class_name] = {
                            "file_path": file_path,
                            "methods": methods,
                        }
                except Exception as e:
                    print(f"Error parsing {file_path}: {e}")
        return class_map

    def validate_executor_methods(self, executor_methods_path: str =
"src/saaaaaa/core/orchestrator/executors_methods.json") -> Dict[str, List[str]]:
        with open(executor_methods_path, "r") as f:
            executor_data = json.load(f)

        declared_methods = set()
        for executor_info in executor_data:
            for method_info in executor_info.get("methods", []):
                class_name = method_info.get("class")
                method_name = method_info.get("method")
                if class_name and method_name:
                    declared_methods.add(f"{class_name}.{method_name}")

        valid = []
        missing = []

        for method_fqn in declared_methods:
            if "." not in method_fqn:
                # Assuming methods are always Class.method
                continue

            class_name, method_name = method_fqn.split(".", 1)

            if class_name not in self.source_map:
                missing.append(method_fqn)
                continue

            class_info = self.source_map[class_name]
            if method_name not in class_info["methods"]:
                missing.append(method_fqn)
            else:
                valid.append(method_fqn)

        # Phantom methods would be those in source but not declared.
        # The user's request seems to focus on missing/valid from declaration.
        # "phantom" is defined by user as "executors call fantasy methods"
        # which is covered by "missing"
        return {"valid": valid, "missing": missing, "phantom": []}


    def generate_source_truth_map(self) -> Dict[str, Dict[str, Any]]:
        source_truth = {}
        for class_name, info in self.source_map.items():
            file_path = info["file_path"]
            with open(file_path, "r", encoding="utf-8") as f:
                tree = ast.parse(f.read(), filename=file_path)
                for node in ast.walk(tree):
                    if isinstance(node, ast.ClassDef) and node.name == class_name:
```

```
                    for item in node.body:
                        if isinstance(item, ast.FunctionDef):
                            method_name = item.name
                            fqn = f"{class_name}.{method_name}"

                            # Basic signature extraction
                            args = [arg.arg for arg in item.args.args]
                            signature = f"({', '.join(args)})"
                            # A more advanced version would parse type hints if they
exist

                            source_truth[fqn] = {
                                "exists": True,
                                "file": file_path,
                                "line": item.lineno,
                                "signature": signature,
                            }
        return source_truth

if __name__ == "__main__":
    validator = MethodSourceValidator()

    # 1. Generate the ground-truth map
    source_truth_map = validator.generate_source_truth_map()
    output_path = "method_source_truth.json"
    with open(output_path, "w") as f:
        json.dump(source_truth_map, f, indent=4)
    print(f"Generated source truth map at {output_path}")

    # 2. Validate executor methods
    validation_report = validator.validate_executor_methods()
    report_path = "executor_validation_report.json"
    with open(report_path, "w") as f:
        json.dump(validation_report, f, indent=4)
    print(f"Validation report generated at {report_path}")

    print("\nValidation Summary:")
    print(f"  - Valid methods: {len(validation_report['valid'])}")
    print(f"  - Missing methods: {len(validation_report['missing'])}")
    if validation_report['missing']:
        print("\nMissing methods:")
        for method in validation_report['missing']:
            print(f"  - {method}")

===== FILE: src/saaaaaa/core/orchestrator/seed_registry.py =====
"""
Seed Registry for Deterministic Execution

Centralized seed management for reproducible stochastic operations across
the orchestrator and all executors.

Key Features:
- SHA256-based seed derivation from policy_unit_id + correlation_id + component
- Unique seeds per component (numpy, python, quantum, neuromorphic, meta-learner)
- Version tracking for seed generation algorithm
- Audit trail for debugging non-determinism
"""

from __future__ import annotations

import hashlib
import logging
from dataclasses import dataclass, field
from datetime import datetime

logger = logging.getLogger(__name__)

# Current seed derivation algorithm version
```

```
SEED_VERSION = "sha256_v1"


@dataclass
class SeedRecord:
    """Record of a generated seed for audit purposes."""
    policy_unit_id: str
    correlation_id: str
    component: str
    seed: int
    timestamp: datetime = field(default_factory=datetime.utcnow)
    seed_version: str = SEED_VERSION


class SeedRegistry:
    """
    Central registry for deterministic seed generation and tracking.

    Ensures that all stochastic operations (NumPy RNG, Python random, quantum
    optimizers, neuromorphic controllers, meta-learner strategies) receive
    consistent, reproducible seeds derived from execution context.

    Usage:
        registry = SeedRegistry()
        np_seed = registry.get_seed(
            policy_unit_id="plan_2024",
            correlation_id="exec_12345",
            component="numpy"
        )
        rng = np.random.default_rng(np_seed)
    """

    def __init__(self) -> None:
        """Initialize seed registry with empty audit log."""
        self._audit_log: list[SeedRecord] = []
        self._seed_cache: dict[tuple[str, str, str], int] = {}
        logger.info(f"SeedRegistry initialized with version {SEED_VERSION}")

    def get_seed(
        self,
        policy_unit_id: str,
        correlation_id: str,
        component: str
    ) -> int:
        """
        Get deterministic seed for a specific component.

        Args:
            policy_unit_id: Unique identifier for the policy document/unit
            correlation_id: Unique identifier for this execution context
            component: Component name (numpy, python, quantum, neuromorphic, meta_learner)

        Returns:
            Deterministic 32-bit unsigned integer seed

        Examples:
            >>> registry = SeedRegistry()
            >>> seed1 = registry.get_seed("plan_2024", "exec_001", "numpy")
            >>> seed2 = registry.get_seed("plan_2024", "exec_001", "numpy")
            >>> assert seed1 == seed2  # Same inputs = same seed
        """
        # Check cache first
        cache_key = (policy_unit_id, correlation_id, component)
        if cache_key in self._seed_cache:
            return self._seed_cache[cache_key]

        # Derive seed
        base_material = f"{policy_unit_id}:{correlation_id}:{component}"
```

```
    seed = self.derive_seed(base_material)

    # Cache and audit
    self._seed_cache[cache_key] = seed
    self._audit_log.append(SeedRecord(
        policy_unit_id=policy_unit_id,
        correlation_id=correlation_id,
        component=component,
        seed=seed
    ))

    logger.debug(
        f"Generated seed {seed} for component={component}, "
        f"policy_unit_id={policy_unit_id}, correlation_id={correlation_id}"
    )

    return seed

def derive_seed(self, base_material: str) -> int:
    """
    Derive deterministic seed from base material using SHA256.

    Args:
        base_material: String to hash (e.g., "plan_2024:exec_001:numpy")

    Returns:
        32-bit unsigned integer seed derived from hash

    Implementation:
        - Uses SHA256 for cryptographic strength
        - Takes first 4 bytes of digest
        - Converts to unsigned 32-bit integer
        - Ensures seed fits in range [0, 2^32-1]
    """
    digest = hashlib.sha256(base_material.encode("utf-8")).digest()
    seed = int.from_bytes(digest[:4], byteorder="big")
    return seed

def get_audit_log(self) -> list[SeedRecord]:
    """
    Get complete audit log of all generated seeds.
```