

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 14
3: =====
4: Generated: 2025-12-07T06:17:21.259895
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfán_pipeline/core/orchestrator/evidence_assembler.py
11: =====
12:
13: from __future__ import annotations
14:
15: import statistics
16: from typing import Any, Iterable, Literal
17:
18: try:
19:     import structlog
20:     logger = structlog.get_logger(__name__)
21: except ImportError:
22:     import logging
23:     logger = logging.getLogger(__name__)
24:
25: def _resolve_value(source: str, method_outputs: dict[str, Any]) -> Any:
26:     """Resolve dotted source paths from method_outputs."""
27:     if not source:
28:         return None
29:     parts = source.split(".")
30:     current: Any = method_outputs
31:     for idx, part in enumerate(parts):
32:         if idx == 0 and part in method_outputs:
33:             current = method_outputs[part]
34:             continue
35:         if isinstance(current, dict) and part in current:
36:             current = current[part]
37:         else:
38:             return None
39:     return current
40:
41:
42: class EvidenceAssembler:
43:     """
44:         Assemble evidence fields from method outputs using deterministic merge strategies.
45:     """
46:
47:     MERGE_STRATEGIES = {
48:         "concat",
49:         "first",
50:         "last",
51:         "mean",
52:         "max",
53:         "min",
54:         "weighted_mean",
55:         "majority",
56:     }
```

```
57:
58:     @staticmethod
59:     def assemble(
60:         method_outputs: dict[str, Any],
61:         assembly_rules: list[dict[str, Any]],
62:         signal_pack: Any | None = None, # NEW: Optional signal pack for provenance
63:     ) -> dict[str, Any]:
64:         evidence: dict[str, Any] = {}
65:         trace: dict[str, Any] = {}
66:
67:         # NEW: Track signal pack provenance if provided
68:         if signal_pack is not None:
69:             trace["signal_provenance"] = {
70:                 "signal_pack_id": getattr(signal_pack, "id", None) or getattr(signal_pack, "pack_id", "unknown"),
71:                 "policy_area": getattr(signal_pack, "policy_area", None) or getattr(signal_pack, "policy_area_id", None),
72:                 "version": getattr(signal_pack, "version", "unknown"),
73:                 "patterns_available": len(getattr(signal_pack, "patterns", [])),
74:                 "source_hash": getattr(signal_pack, "source_hash", None),
75:             }
76:             logger.info(
77:                 "signal_pack_attached",
78:                 signal_pack_id=trace["signal_provenance"]["signal_pack_id"],
79:                 policy_area=trace["signal_provenance"]["policy_area"],
80:             )
81:
82:         if "_signal_usage" in method_outputs:
83:             logger.info("signal_consumption_trace", signals_used=method_outputs["_signal_usage"])
84:             trace["signal_usage"] = method_outputs["_signal_usage"]
85:             # Remove from method_outputs to not interfere with evidence assembly
86:             del method_outputs["_signal_usage"]
87:
88:         for rule in assembly_rules:
89:             target = rule.get("target")
90:             sources: Iterable[str] = rule.get("sources", [])
91:             strategy: str = rule.get("merge_strategy", "first")
92:             weights: list[float] | None = rule.get("weights")
93:             default = rule.get("default")
94:
95:             if strategy not in EvidenceAssembler.MERGE_STRATEGIES:
96:                 raise ValueError(f"Unsupported merge_strategy '{strategy}' for target '{target}'")
97:
98:             values = []
99:             for src in sources:
100:                 val = _resolve_value(src, method_outputs)
101:                 if val is not None:
102:                     values.append(val)
103:
104:             merged = EvidenceAssembler._merge(values, strategy, weights, default)
105:             evidence[target] = merged
106:             trace[target] = {"sources": list(sources), "strategy": strategy, "values": values}
107:
108:         return {"evidence": evidence, "trace": trace}
109:
110:     @staticmethod
111:     def _merge(values: list[Any], strategy: str, weights: list[float] | None, default: Any) -> Any:
112:         if not values:
```

```
113:         return default
114:     if strategy == "first":
115:         return values[0]
116:     if strategy == "last":
117:         return values[-1]
118:     if strategy == "concat":
119:         merged: list[Any] = []
120:         for v in values:
121:             if isinstance(v, list):
122:                 merged.extend(v)
123:             else:
124:                 merged.append(v)
125:         return merged
126:     numeric_values = [float(v) for v in values if EvidenceAssembler._is_number(v)]
127:     if strategy == "mean":
128:         return statistics.fmean(numeric_values) if numeric_values else default
129:     if strategy == "max":
130:         return max(numeric_values) if numeric_values else default
131:     if strategy == "min":
132:         return min(numeric_values) if numeric_values else default
133:     if strategy == "weighted_mean":
134:         if not numeric_values:
135:             return default
136:         if not weights:
137:             weights = [1.0] * len(numeric_values)
138:         w = weights[: len(numeric_values)] or [1.0] * len(numeric_values)
139:         total = sum(w) or 1.0
140:         return sum(v * w_i for v, w_i in zip(numeric_values, w)) / total
141:     if strategy == "majority":
142:         counts: dict[Any, int] = {}
143:         for v in values:
144:             counts[v] = counts.get(v, 0) + 1
145:         return max(counts.items(), key=lambda item: item[1])[0] if counts else default
146:     return default
147:
148: @staticmethod
149: def _is_number(value: Any) -> bool:
150:     try:
151:         float(value)
152:         return not isinstance(value, bool)
153:     except (TypeError, ValueError) as e:
154:         if logger.isEnabledFor(logging.DEBUG):
155:             logger.debug(f"Non-numeric value: {value!r} ({type(value).__name__}): {e}")
156:         return False
157:
158:
159:
160: =====
161: FILE: src/farfan_pipeline/core/orchestrator/evidence_registry.py
162: =====
163:
164: """
165: Evidence Registry: Append-Only JSONL Store with Hash Chain and Provenance DAG Export
166:
167: This module implements a comprehensive evidence tracking system that:
168: 1. Stores all evidence in append-only JSONL format for immutability
```

```
169: 2. Maintains hash-based indexing for fast evidence lookup
170: 3. Implements blockchain-style hash chaining for ledger integrity
171: 4. Exports provenance DAG showing evidence lineage and dependencies
172: 5. Provides cryptographic verification of evidence integrity
173:
174: Architecture:
175: - JSONL Storage: One JSON object per line, append-only for audit trail
176: - Hash Index: SHA-256 hashes for content-addressable storage
177: - Hash Chain: Each entry links to previous via previous_hash and entry_hash
178: - Provenance DAG: Directed acyclic graph of evidence dependencies
179: - Verification: Cryptographic chain-of-custody validation with chain linkage checks
180:
181: Hash Chain Security:
182: The registry implements a blockchain-style hash chain where each entry contains:
183: - content_hash: SHA-256 of the payload (for content verification)
184: - previous_hash: Hash of the previous entry's entry_hash (creates the chain)
185: - entry_hash: SHA-256 of (content_hash + previous_hash + metadata)
186:
187: This ensures that:
188: 1. Any tampering with payload is detected via content_hash mismatch
189: 2. Any tampering with previous_hash is detected via chain verification
190: 3. Entries cannot be reordered without breaking the chain
191: 4. The entire ledger history can be cryptographically verified
192: """
193:
194: from __future__ import annotations
195:
196: import hashlib
197: import json
198: import logging
199: import time
200: from collections import defaultdict
201: from dataclasses import asdict, dataclass, field
202: from datetime import datetime, timezone
203: from pathlib import Path
204: from typing import Any
205:
206: logger = logging.getLogger(__name__)
207:
208: @dataclass
209: class EvidenceRecord:
210:     """
211:         Immutable evidence record with provenance metadata and hash chain linkage.
212:
213:         Each evidence record captures:
214:             - Unique identifier (hash-based)
215:             - Evidence payload (method result, analysis output, etc.)
216:             - Provenance metadata (source, dependencies, lineage)
217:             - Temporal metadata (timestamp, execution time)
218:             - Verification data (content hash, chain hashes)
219:
220:         Hash Chain Fields:
221:             - content_hash: SHA-256 of payload (verifies content integrity)
222:             - previous_hash: entry_hash of previous record (creates chain linkage)
223:             - entry_hash: SHA-256 of (content + previous_hash + metadata) (unique entry ID)
224:
```

```
225:     The hash chain ensures that:
226:     1. Tampering with payload breaks content_hash
227:     2. Tampering with previous_hash breaks chain verification
228:     3. Entire ledger history is cryptographically verifiable
229: """
230:
231: # Identification
232: evidence_id: str # SHA-256 hash of content
233: evidence_type: str # Type of evidence (e.g., "method_result", "analysis", "extraction")
234:
235: # Payload
236: payload: dict[str, Any]
237:
238: # Provenance
239: source_method: str | None = None # FQN of method that produced this evidence
240: parent_evidence_ids: list[str] = field(default_factory=list) # Dependencies
241: question_id: str | None = None
242: document_id: str | None = None
243:
244: # Temporal
245: timestamp: float = field(default_factory=time.time)
246: execution_time_ms: float = 0.0
247:
248: # Verification
249: content_hash: str | None = None # Hash of payload for verification
250: previous_hash: str | None = None # Hash of previous entry in chain (for ledger integrity)
251: entry_hash: str | None = None # Hash of this entire entry including previous_hash
252:
253: # Metadata
254: metadata: dict[str, Any] = field(default_factory=dict)
255:
256: def __post_init__(self):
257:     """Generate content hash and entry hash if not provided."""
258:     if self.content_hash is None:
259:         self.content_hash = self._compute_content_hash()
260:     if self.entry_hash is None:
261:         self.entry_hash = self._compute_entry_hash()
262:
263: def _canonical_dump(self, obj: Any) -> str:
264: """
265:     Create canonical JSON representation for deterministic hashing.
266:
267:     This method ensures:
268:     - Keys are sorted alphabetically
269:     - No whitespace in output
270:     - Consistent handling of None, booleans, numbers
271:     - Deterministic ordering for nested structures
272:     - Unicode normalization
273:
274:     Uses a custom JSON serialization handler to support non-standard types
275:     commonly found in evidence payloads (dataclasses, NumPy arrays, custom objects).
276:     The handler converts objects to dicts via __dict__ or falls back to string
277:     representation, ensuring all evidence can be serialized without exceptions.
278:
279:     Args:
280:         obj: Object to serialize
```

```
281:
282:     Returns:
283:         Canonical JSON string
284:     """
285:     # Use separators with no spaces and sort keys for determinism
286:     # ensure_ascii=True ensures consistent output across platforms
287:     # Custom handler for non-serializable types (dataclasses, NumPy arrays, etc.)
288:     def default_handler(o):
289:         if hasattr(o, '__dict__'):
290:             return o.__dict__
291:         return str(o)
292:
293:     return json.dumps(
294:         obj,
295:         sort_keys=True,
296:         separators=(',', ','),
297:         ensure_ascii=True,
298:         default=default_handler
299:     )
300:
301: def _compute_content_hash(self) -> str:
302:     """
303:     Compute SHA-256 hash of payload for content-addressable storage.
304:
305:     Uses canonical JSON serialization to ensure deterministic hashing
306:     across different Python versions and platforms.
307:
308:     Returns:
309:         Hex digest of SHA-256 hash
310:     """
311:     # Create deterministic JSON representation using canonical dump
312:     payload_json = self._canonical_dump(self.payload)
313:
314:     # Compute SHA-256 hash
315:     hash_obj = hashlib.sha256(payload_json.encode('utf-8'))
316:     return hash_obj.hexdigest()
317:
318: def _compute_entry_hash(self) -> str:
319:     """
320:     Compute SHA-256 hash of the entire entry including previous_hash.
321:     This creates the hash chain linking entries together.
322:
323:     Uses canonical JSON serialization for deterministic hashing.
324:
325:     Returns:
326:         Hex digest of SHA-256 hash
327:     """
328:     # Combine content hash with previous hash to create chain
329:     # Use empty string for first entry (no predecessor)
330:     chain_data = {
331:         'content_hash': self.content_hash,
332:         'previous_hash': self.previous_hash if self.previous_hash is not None else '',
333:         'evidence_type': self.evidence_type,
334:         'timestamp': self.timestamp,
335:     }
336:     chain_json = self._canonical_dump(chain_data)
```

```
337:     hash_obj = hashlib.sha256(chain_json.encode('utf-8'))
338:     return hash_obj.hexdigest()
339:
340:     def verify_integrity(self, previous_record: EvidenceRecord | None = None) -> bool:
341:         """
342:             Verify evidence integrity by recomputing hashes and checking chain linkage.
343:
344:             Args:
345:                 previous_record: The record that should precede this one in the chain
346:
347:             Returns:
348:                 True if all integrity checks pass, False otherwise
349:             """
350:             # Verify content hash matches
351:             current_content_hash = self._compute_content_hash()
352:             if current_content_hash != self.content_hash:
353:                 return False
354:
355:             # Verify entry hash matches
356:             current_entry_hash = self._compute_entry_hash()
357:             if current_entry_hash != self.entry_hash:
358:                 return False
359:
360:             # If previous record is provided, verify the chain linkage
361:             if previous_record is not None:
362:                 # Verify that our previous_hash matches the actual hash of the previous record
363:                 if self.previous_hash != previous_record.entry_hash:
364:                     return False
365:
366:             return True
367:
368:     def to_dict(self) -> dict[str, Any]:
369:         """Convert to dictionary for serialization."""
370:         return asdict(self)
371:
372:     @classmethod
373:     def from_dict(cls, data: dict[str, Any]) -> EvidenceRecord:
374:         """Create evidence record from dictionary."""
375:         return cls(**data)
376:
377:     @classmethod
378:     def create(
379:         cls,
380:         evidence_type: str,
381:         payload: dict[str, Any],
382:         source_method: str | None = None,
383:         parent_evidence_ids: list[str] | None = None,
384:         question_id: str | None = None,
385:         document_id: str | None = None,
386:         execution_time_ms: float = 0.0,
387:         metadata: dict[str, Any] | None = None,
388:         previous_hash: str | None = None,
389:     ) -> EvidenceRecord:
390:         """
391:             Create a new evidence record with proper hash computation.
392:
```

```
393:     This factory method ensures:
394:     - Proper initialization order
395:     - Deterministic hash computation
396:     - Validation of required fields
397:
398:     Args:
399:         evidence_type: Type of evidence
400:         payload: Evidence data (must be JSON-serializable)
401:         source_method: FQN of method that produced evidence
402:         parent_evidence_ids: List of parent evidence IDs
403:         question_id: Question ID this evidence relates to
404:         document_id: Document ID this evidence relates to
405:         execution_time_ms: Execution time in milliseconds
406:         metadata: Additional metadata
407:         previous_hash: Hash of previous entry in chain (for chain linkage)
408:
409:     Returns:
410:         New EvidenceRecord instance
411:
412:     Raises:
413:         ValueError: If required fields are invalid or payload is not serializable
414:         """
415:     if not evidence_type:
416:         raise ValueError("evidence_type is required")
417:
418:     if not isinstance(payload, dict):
419:         raise ValueError("payload must be a dictionary")
420:
421:     # Test that payload is JSON-serializable
422:     try:
423:         json.dumps(payload)
424:     except (TypeError, ValueError) as e:
425:         raise ValueError(f"payload must be JSON-serializable: {e}")
426:
427:     # Create record with temporary evidence_id
428:     record = cls(
429:         evidence_id="",
430:         evidence_type=evidence_type,
431:         payload=payload,
432:         source_method=source_method,
433:         parent_evidence_ids=parent_evidence_ids or [],
434:         question_id=question_id,
435:         document_id=document_id,
436:         execution_time_ms=execution_time_ms,
437:         metadata=metadata or {},
438:         previous_hash=previous_hash,
439:     )
440:
441:     # Set evidence_id to content hash (computed in __post_init__)
442:     record.evidence_id = record.content_hash or ""
443:
444:     return record
445:
446: @dataclass
447: class ProvenanceNode:
448:     """Node in provenance DAG."""

```

```
449:
450:     evidence_id: str
451:     evidence_type: str
452:     source_method: str | None
453:     timestamp: float
454:     children: list[str] = field(default_factory=list) # Evidence IDs that depend on this
455:     parents: list[str] = field(default_factory=list) # Evidence IDs this depends on
456:
457:     def to_dict(self) -> dict[str, Any]:
458:         """Convert to dictionary."""
459:         return asdict(self)
460:
461: @dataclass
462: class ProvenanceDAG:
463:     """
464:         Directed Acyclic Graph of evidence provenance.
465:
466:         Captures the full lineage of evidence:
467:             - Which evidence produced which other evidence
468:             - Method invocation chains
469:             - Data flow dependencies
470:     """
471:
472:     nodes: dict[str, ProvenanceNode] = field(default_factory=dict)
473:
474:     # Index for fast queries
475:     by_method: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
476:     by_type: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
477:     by_question: dict[str, list[str]] = field(default_factory=lambda: defaultdict(list))
478:
479:     def add_evidence(
480:         self,
481:         evidence: EvidenceRecord
482:     ) -> None:
483:         """
484:             Add evidence to provenance DAG.
485:
486:             Args:
487:                 evidence: Evidence record to add
488:         """
489:         # Create node
490:         node = ProvenanceNode(
491:             evidence_id=evidence.evidence_id,
492:             evidence_type=evidence.evidence_type,
493:             source_method=evidence.source_method,
494:             timestamp=evidence.timestamp,
495:             parents=evidence.parent_evidence_ids.copy(),
496:         )
497:
498:         # Add to nodes
499:         self.nodes[evidence.evidence_id] = node
500:
501:         # Update parent-child relationships
502:         for parent_id in evidence.parent_evidence_ids:
503:             if parent_id in self.nodes:
504:                 self.nodes[parent_id].children.append(evidence.evidence_id)
```

```
505:  
506:     # Update indices  
507:     if evidence.source_method:  
508:         self.by_method[evidence.source_method].append(evidence.evidence_id)  
509:     self.by_type[evidence.evidence_type].append(evidence.evidence_id)  
510:     if evidence.question_id:  
511:         self.by_question[evidence.question_id].append(evidence.evidence_id)  
512:  
513:     def get_ancestors(self, evidence_id: str) -> set[str]:  
514:         """  
515:             Get all ancestor evidence IDs (transitive parents).  
516:  
517:             Args:  
518:                 evidence_id: Evidence ID to trace  
519:  
520:             Returns:  
521:                 Set of ancestor evidence IDs  
522:         """  
523:         ancestors = set()  
524:         visited = set()  
525:  
526:         def traverse(eid: str) -> None:  
527:             if eid in visited:  
528:                 return  
529:             visited.add(eid)  
530:  
531:             if eid not in self.nodes:  
532:                 return  
533:  
534:             node = self.nodes[eid]  
535:             for parent_id in node.parents:  
536:                 ancestors.add(parent_id)  
537:                 traverse(parent_id)  
538:  
539:         traverse(evidence_id)  
540:         return ancestors  
541:  
542:     def get_descendants(self, evidence_id: str) -> set[str]:  
543:         """  
544:             Get all descendant evidence IDs (transitive children).  
545:  
546:             Args:  
547:                 evidence_id: Evidence ID to trace  
548:  
549:             Returns:  
550:                 Set of descendant evidence IDs  
551:         """  
552:         descendants = set()  
553:         visited = set()  
554:  
555:         def traverse(eid: str) -> None:  
556:             if eid in visited:  
557:                 return  
558:             visited.add(eid)  
559:  
560:             if eid not in self.nodes:
```

```
561:         return
562:
563:     node = self.nodes[eid]
564:     for child_id in node.children:
565:         descendants.add(child_id)
566:         traverse(child_id)
567:
568:     traverse(evidence_id)
569:     return descendants
570:
571: def get_lineage(self, evidence_id: str) -> dict[str, Any]:
572:     """
573:     Get complete lineage for evidence (ancestors + descendants).
574:
575:     Args:
576:         evidence_id: Evidence ID to trace
577:
578:     Returns:
579:         Dictionary with lineage information
580:     """
581:     return {
582:         "evidence_id": evidence_id,
583:         "ancestors": list(self.get_ancestors(evidence_id)),
584:         "descendants": list(self.get_descendants(evidence_id)),
585:         "ancestor_count": len(self.get_ancestors(evidence_id)),
586:         "descendant_count": len(self.get_descendants(evidence_id)),
587:     }
588:
589: def export_dot(self) -> str:
590:     """
591:     Export DAG in GraphViz DOT format.
592:
593:     Returns:
594:         DOT format string
595:     """
596:     lines = ["digraph ProvenanceDAG {"]
597:     lines.append("    rankdir=LR;")
598:     lines.append("    node [shape=box];")
599:     lines.append("")
600:
601:     # Add nodes
602:     for eid, node in self.nodes.items():
603:         label = f"{node.evidence_type}\n{eid[:8]}..."
604:         if node.source_method:
605:             label += f"\n{node.source_method}"
606:         lines.append(f'    "{eid}" [label="{label}"];')
607:
608:     lines.append("")
609:
610:     # Add edges
611:     for eid, node in self.nodes.items():
612:         for child_id in node.children:
613:             lines.append(f'    "{eid}" -> "{child_id}";')
614:
615:     lines.append("}")
616:     return "\n".join(lines)
```

```
617:
618:     def to_dict(self) -> dict[str, Any]:
619:         """Export DAG to dictionary."""
620:         return {
621:             "nodes": {eid: node.to_dict() for eid, node in self.nodes.items()},
622:             "stats": {
623:                 "total_nodes": len(self.nodes),
624:                 "by_method": {k: len(v) for k, v in self.by_method.items()},
625:                 "by_type": {k: len(v) for k, v in self.by_type.items()},
626:                 "by_question": {k: len(v) for k, v in self.by_question.items()},
627:             }
628:         }
629:
630: class EvidenceRegistry:
631:     """
632:         Append-only evidence registry with hash indexing and provenance tracking.
633:
634:     Features:
635:         - JSONL append-only storage for immutability
636:         - Content-addressable hash indexing
637:         - Provenance DAG for lineage tracking
638:         - Cryptographic verification
639:         - Fast queries by hash, type, method, question
640:
641:     """
642:     def __init__(
643:         self,
644:         storage_path: Path | None = None,
645:         enable_dag: bool = True,
646:     ) -> None:
647:         """
648:             Initialize evidence registry.
649:
650:             Args:
651:                 storage_path: Path to JSONL storage file (default: evidence_registry.jsonl)
652:                 enable_dag: Enable provenance DAG tracking
653:             """
654:         self.storage_path = storage_path or Path("evidence_registry.jsonl")
655:         self.enable_dag = enable_dag
656:
657:         # Hash index: hash -> evidence record
658:         self.hash_index: dict[str, EvidenceRecord] = {}
659:
660:         # Type index: type -> list of hashes
661:         self.type_index: dict[str, list[str]] = defaultdict(list)
662:
663:         # Method index: method FQN -> list of hashes
664:         self.method_index: dict[str, list[str]] = defaultdict(list)
665:
666:         # Question index: question ID -> list of hashes
667:         self.question_index: dict[str, list[str]] = defaultdict(list)
668:
669:         # Provenance DAG
670:         self.dag = ProvenanceDAG() if enable_dag else None
671:
672:         # Track the last entry in the ledger chain for hash chaining
```

```
673:         self.last_entry: EvidenceRecord | None = None
674:
675:     # Load existing evidence
676:     self._load_from_storage()
677:
678:     logger.info(
679:         f"EvidenceRegistry initialized with {len(self.hash_index)} records, "
680:         f"storage={self.storage_path}, dag={'enabled' if enable_dag else 'disabled'}"
681:     )
682:
683:     def _load_from_storage(self) -> None:
684:         """
685:             Load evidence from JSONL storage with chain verification.
686:
687:             Ensures:
688:                 - Evidence is loaded in the order it was written
689:                 - Chain linkage is validated during load
690:                 - Index ordering is preserved
691:         """
692:         if not self.storage_path.exists():
693:             logger.info(f"No existing evidence storage found at {self.storage_path}")
694:             return
695:
696:         loaded_count = 0
697:         BATCH_SIZE = 1000
698:         batch_records = []
699:
700:         try:
701:             with open(self.storage_path, encoding='utf-8') as f:
702:                 for line_num, line in enumerate(f, 1):
703:                     try:
704:                         data = json.loads(line.strip())
705:                         evidence = EvidenceRecord.from_dict(data)
706:                         batch_records.append((line_num, evidence))
707:                         loaded_count += 1
708:
709:                         if len(batch_records) >= BATCH_SIZE:
710:                             self._assert_chain(batch_records)
711:                             for _, ev in batch_records:
712:                                 self._index_evidence(ev, persist=False)
713:                             batch_records.clear()
714:
715:                         except json.JSONDecodeError as e:
716:                             logger.warning(f"Failed to parse line {line_num}: {e}")
717:                         except Exception as e:
718:                             logger.warning(f"Failed to load evidence on line {line_num}: {e}")
719:
720:                         if batch_records:
721:                             self._assert_chain(batch_records)
722:                             for _, ev in batch_records:
723:                                 self._index_evidence(ev, persist=False)
724:                             batch_records.clear()
725:
726:                         logger.info(f"Loaded {loaded_count} evidence records from storage")
727:
728:             except Exception as e:
```

```
729:         logger.error(f"Failed to load evidence storage: {e}")
730:
731:     def _assert_chain(self, records: list[tuple[int, EvidenceRecord]]) -> None:
732:         """
733:             Assert that the chain of evidence records is valid.
734:
735:             Validates:
736:                 - First record has no previous_hash or previous_hash is None
737:                 - Each subsequent record's previous_hash matches the prior record's entry_hash
738:                 - Records are in the correct sequential order
739:
740:             Args:
741:                 records: List of (line_number, EvidenceRecord) tuples in load order
742:
743:             Raises:
744:                 ValueError: If chain validation fails
745:             """
746:         if not records:
747:             return
748:
749:         previous_record = None
750:
751:         for idx, (line_num, record) in enumerate(records):
752:             if idx == 0:
753:                 # First record should have no previous_hash or None
754:                 if record.previous_hash and record.previous_hash != '':
755:                     logger.warning(
756:                         f"Line {line_num}: First record has previous_hash={record.previous_hash}, "
757:                         f"expected None or empty string. Chain may have been corrupted or truncated."
758:                     )
759:             # Subsequent records should link to the previous record
760:             elif previous_record is not None:
761:                 expected_previous_hash = previous_record.entry_hash
762:                 actual_previous_hash = record.previous_hash
763:
764:                 if actual_previous_hash != expected_previous_hash:
765:                     raise ValueError(
766:                         f"Chain broken at line {line_num}: "
767:                         f"expected previous_hash={expected_previous_hash}, "
768:                         f"got previous_hash={actual_previous_hash}. "
769:                         f"Evidence ordering may be corrupted."
770:                     )
771:
772:             previous_record = record
773:
774:     def _index_evidence(
775:         self,
776:         evidence: EvidenceRecord,
777:         persist: bool = True
778:     ) -> None:
779:         """
780:             Index evidence record in all indices.
781:
782:             Args:
783:                 evidence: Evidence to index
784:                 persist: If True, append to JSONL storage
```

```
785:     """
786:     # Hash index
787:     self.hash_index[evidence.evidence_id] = evidence
788:
789:     # Type index
790:     self.type_index[evidence.evidence_type].append(evidence.evidence_id)
791:
792:     # Method index
793:     if evidence.source_method:
794:         self.method_index[evidence.source_method].append(evidence.evidence_id)
795:
796:     # Question index
797:     if evidence.question_id:
798:         self.question_index[evidence.question_id].append(evidence.evidence_id)
799:
800:     # DAG
801:     if self.enable_dag and self.dag:
802:         self.dag.add_evidence(evidence)
803:
804:     # Update last entry for hash chaining
805:     self.last_entry = evidence
806:
807:     # Persist to storage
808:     if persist:
809:         self._append_to_storage(evidence)
810:
811:     def _append_to_storage(self, evidence: EvidenceRecord) -> None:
812:         """
813:             Append evidence to JSONL storage.
814:
815:             Args:
816:                 evidence: Evidence to append
817:             """
818:             try:
819:                 # Ensure parent directory exists
820:                 self.storage_path.parent.mkdir(parents=True, exist_ok=True)
821:
822:                 # Append to JSONL
823:                 with open(self.storage_path, 'a', encoding='utf-8') as f:
824:                     json_line = json.dumps(evidence.to_dict(), separators=(',', ':'))
825:                     f.write(json_line + '\n')
826:
827:             except Exception as e:
828:                 logger.error(f"Failed to append evidence to storage: {e}")
829:                 raise
830:
831:     def record_evidence(
832:         self,
833:         evidence_type: str,
834:         payload: dict[str, Any],
835:         source_method: str | None = None,
836:         parent_evidence_ids: list[str] | None = None,
837:         question_id: str | None = None,
838:         document_id: str | None = None,
839:         execution_time_ms: float = 0.0,
840:         metadata: dict[str, Any] | None = None,
```

```
841:     ) -> str:
842:         """
843:             Record new evidence in registry.
844:
845:             Args:
846:                 evidence_type: Type of evidence
847:                 payload: Evidence data
848:                 source_method: FQN of method that produced evidence
849:                 parent_evidence_ids: List of parent evidence IDs
850:                 question_id: Question ID this evidence relates to
851:                 document_id: Document ID this evidence relates to
852:                 execution_time_ms: Execution time
853:                 metadata: Additional metadata
854:
855:             Returns:
856:                 Evidence ID (hash)
857:             """
858:             # Determine previous_hash from last entry in the chain
859:             previous_hash = self.last_entry.entry_hash if self.last_entry else None
860:
861:             # Normalize metadata and ensure recorded_at timestamp
862:             metadata_dict: dict[str, Any] = dict(metadata) if metadata else {}
863:             metadata_dict.setdefault(
864:                 "recorded_at",
865:                 datetime.now(timezone.utc).isoformat(),
866:             )
867:
868:             # Create evidence record with deterministic hash-based ID
869:             evidence = EvidenceRecord.create(
870:                 evidence_type=evidence_type,
871:                 payload=payload,
872:                 source_method=source_method,
873:                 parent_evidence_ids=parent_evidence_ids,
874:                 question_id=question_id,
875:                 document_id=document_id,
876:                 execution_time_ms=execution_time_ms,
877:                 metadata=metadata_dict,
878:                 previous_hash=previous_hash,
879:             )
880:
881:             # Check for duplicate
882:             if evidence.evidence_id in self.hash_index:
883:                 logger.debug(f"Evidence {evidence.evidence_id} already exists, skipping")
884:                 return evidence.evidence_id
885:
886:             # Index evidence
887:             self._index_evidence(evidence, persist=True)
888:
889:             logger.debug(f"Recorded evidence {evidence.evidence_id} of type {evidence_type}")
890:
891:             return evidence.evidence_id
892:
893:     def get_evidence(self, evidence_id: str) -> EvidenceRecord | None:
894:         """
895:             Retrieve evidence by ID.
896:
```

```
897:     Args:
898:         evidence_id: Evidence hash
899:
900:     Returns:
901:         EvidenceRecord or None
902:     """
903:     return self.hash_index.get(evidence_id)
904:
905: def query_by_type(self, evidence_type: str) -> list[EvidenceRecord]:
906:     """Query evidence by type."""
907:     evidence_ids = self.type_index.get(evidence_type, [])
908:     return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]
909:
910: def query_by_method(self, method_fqn: str) -> list[EvidenceRecord]:
911:     """Query evidence by source method."""
912:     evidence_ids = self.method_index.get(method_fqn, [])
913:     return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]
914:
915: def query_by_question(self, question_id: str) -> list[EvidenceRecord]:
916:     """Query evidence by question ID."""
917:     evidence_ids = self.question_index.get(question_id, [])
918:     return [self.hash_index[eid] for eid in evidence_ids if eid in self.hash_index]
919:
920: def verify_evidence(self, evidence_id: str, verify_chain: bool = True) -> bool:
921:     """
922:     Verify evidence integrity and optionally chain linkage.
923:
924:     Args:
925:         evidence_id: Evidence hash
926:         verify_chain: If True, verify chain linkage with previous entry
927:
928:     Returns:
929:         True if evidence is valid
930:     """
931:     evidence = self.get_evidence(evidence_id)
932:     if evidence is None:
933:         return False
934:
935:     # Get previous record if chain verification is requested
936:     previous_record = None
937:     if verify_chain and evidence.previous_hash:
938:         # Find the record with entry_hash matching our previous_hash
939:         for record in self.hash_index.values():
940:             if record.entry_hash == evidence.previous_hash:
941:                 previous_record = record
942:                 break
943:
944:     return evidence.verify_integrity(previous_record=previous_record)
945:
946: def verify_chain_integrity(self) -> tuple[bool, list[str]]:
947:     """
948:     Verify the integrity of the entire evidence chain.
949:
950:     Returns:
951:         Tuple of (is_valid, list of errors)
952:     """
```

```
953:     errors = []
954:
955:     # Build the chain by reading from storage in order
956:     if not self.storage_path.exists():
957:         return True, [] # Empty chain is valid
958:
959:     try:
960:         previous_record = None
961:         with open(self.storage_path, encoding='utf-8') as f:
962:             for line_num, line in enumerate(f, 1):
963:                 try:
964:                     data = json.loads(line.strip())
965:                     evidence = EvidenceRecord.from_dict(data)
966:
967:                     # Verify the record's integrity
968:                     if not evidence.verify_integrity(previous_record=previous_record):
969:                         if previous_record and evidence.previous_hash != previous_record.entry_hash:
970:                             errors.append(
971:                                 f"Line {line_num}: Chain broken - previous_hash mismatch. "
972:                                 f"Expected {previous_record.entry_hash}, got {evidence.previous_hash}"
973:                             )
974:                         else:
975:                             errors.append(
976:                                 f"Line {line_num}: Hash integrity check failed for evidence {evidence.evidence_id}"
977:                             )
978:
979:                     previous_record = evidence
980:
981:                 except json.JSONDecodeError as e:
982:                     errors.append(f"Line {line_num}: JSON parsing error - {e}")
983:                 except Exception as e:
984:                     errors.append(f"Line {line_num}: Verification error - {e}")
985:
986:             return len(errors) == 0, errors
987:
988:     except Exception as e:
989:         return False, [f"Failed to verify chain: {e}"]
990:
991:     def get_provenance(self, evidence_id: str) -> dict[str, Any] | None:
992:         """
993:             Get provenance information for evidence.
994:
995:             Args:
996:                 evidence_id: Evidence hash
997:
998:             Returns:
999:                 Provenance dictionary or None
1000: """
1001:     if not self.enable_dag or self.dag is None:
1002:         return None
1003:
1004:     return self.dag.get_lineage(evidence_id)
1005:
1006:     def export_provenance_dag(
1007:         self,
1008:         format: str = "dict",
```

```
1009:         output_path: Path | None = None
1010:     ) -> Any:
1011:     """
1012:     Export provenance DAG.
1013:
1014:     Args:
1015:         format: Export format ("dict", "dot", "json")
1016:         output_path: Optional path to write output
1017:
1018:     Returns:
1019:         Exported DAG in requested format
1020:     """
1021:     if not self.enable_dag or self.dag is None:
1022:         raise ValueError("DAG tracking is not enabled")
1023:
1024:     if format == "dot":
1025:         result = self.dag.export_dot()
1026:     elif format == "dict":
1027:         result = self.dag.to_dict()
1028:     elif format == "json":
1029:         result = json.dumps(self.dag.to_dict(), indent=2)
1030:     else:
1031:         raise ValueError(f"Unsupported format: {format}")
1032:
1033:     # Write to file if path provided
1034:     if output_path:
1035:         output_path.parent.mkdir(parents=True, exist_ok=True)
1036:         if isinstance(result, str):
1037:             output_path.write_text(result, encoding='utf-8')
1038:         else:
1039:             output_path.write_text(json.dumps(result, indent=2), encoding='utf-8')
1040:         logger.info(f"Exported provenance DAG to {output_path}")
1041:
1042:     return result
1043:
1044: def get_statistics(self) -> dict[str, Any]:
1045:     """
1046:     Get registry statistics.
1047:
1048:     Returns:
1049:         Statistics dictionary
1050:     """
1051:     stats = {
1052:         "total_evidence": len(self.hash_index),
1053:         "by_type": {k: len(v) for k, v in self.type_index.items()},
1054:         "by_method": {k: len(v) for k, v in self.method_index.items()},
1055:         "by_question": {k: len(v) for k, v in self.question_index.items()},
1056:         "storage_path": str(self.storage_path),
1057:         "dag_enabled": self.enable_dag,
1058:     }
1059:
1060:     if self.enable_dag and self.dag:
1061:         stats["dag_nodes"] = len(self.dag.nodes)
1062:
1063:     return stats
1064:
```

```
1065:     def stats(self) -> dict[str, int]:
1066:         """Get simplified evidence registry statistics.
1067:
1068:         Returns:
1069:             Dict with counts for records, types, methods, and questions.
1070:         """
1071:
1072:         return {
1073:             "records": len(self.hash_index),
1074:             "types": len(self.type_index),
1075:             "methods": len(self.method_index),
1076:             "questions": len(self.question_index),
1077:         }
1078: # Global registry instance
1079: _global_registry: EvidenceRegistry | None = None
1080:
1081: def get_global_registry() -> EvidenceRegistry:
1082:     """Get or create global evidence registry."""
1083:     global _global_registry
1084:     if _global_registry is None:
1085:         _global_registry = EvidenceRegistry()
1086:     return _global_registry
1087:
1088: __all__ = [
1089:     "EvidenceRecord",
1090:     "ProvenanceNode",
1091:     "ProvenanceDAG",
1092:     "EvidenceRegistry",
1093:     "get_global_registry",
1094: ]
1095:
1096:
1097:
1098: =====
1099: FILE: src/farfan_pipeline/core/orchestrator/evidence_validator.py
1100: =====
1101:
1102: from __future__ import annotations
1103:
1104: import re
1105: from typing import Any, Iterable
1106:
1107:
1108: class EvidenceValidator:
1109:     """Validate assembled evidence with configurable rules."""
1110:
1111:     @staticmethod
1112:     def validate(
1113:         evidence: dict[str, Any],
1114:         rules_object: dict[str, Any],
1115:         failure_contract: dict[str, Any] | None = None, # NEW: Signal-based failure contract
1116:     ) -> dict[str, Any]:
1117:         """
1118:             Validates evidence against a rules object from a V2 contract.
1119:
1120:             Args:
```

```

1121:         evidence: The assembled evidence dictionary.
1122:         rules_object: The validation object from the contract, containing
1123:             'rules' (a list) and 'na_policy' (a string).
1124:         failure_contract: Optional failure contract from signal pack containing
1125:             'abort_if' conditions and 'emit_code' for abort signal.
1126:
1127:     """
1128:     validation_rules = rules_object.get("rules", [])
1129:     na_policy = rules_object.get("na_policy", "abort_on_critical")
1130:     errors: list[str] = []
1131:     warnings: list[str] = []
1132:     abort_code: str | None = None
1133:
1134:     for rule in validation_rules:
1135:         field = rule.get("field")
1136:         value = EvidenceValidator._resolve(field, evidence)
1137:
1138:         # --- New Rich Rule Logic ---
1139:         if rule.get("must_contain"):
1140:             must_contain = rule["must_contain"]
1141:             required_elements = set(must_contain.get("elements", []))
1142:             present_elements = set(value) if isinstance(value, list) else set()
1143:             missing_elements = required_elements - present_elements
1144:             if missing_elements:
1145:                 errors.append(f"Field '{field}' is missing required elements: {', '.join(sorted(missing_elements))}")
1146:
1147:             required_count = must_contain.get("count")
1148:             if required_count and len(present_elements.intersection(required_elements)) < required_count:
1149:                 errors.append(f"Field '{field}' did not meet the required count of {required_count} for elements: {', '.join(sorted(required_elements))}")
1150:
1151:         if rule.get("should_contain"):
1152:             should_contain = rule["should_contain"]
1153:             present_elements = set(value) if isinstance(value, list) else set()
1154:             for requirement in should_contain:
1155:                 elements_to_check = set(requirement.get("elements", []))
1156:                 min_count = requirement.get("minimum", 1)
1157:                 found_count = len(present_elements.intersection(elements_to_check))
1158:                 if found_count < min_count:
1159:                     warnings.append(f"Field '{field}' only has {found_count}/{min_count} of recommended elements: {', '.join(sorted(elements_to_check))}")
1160:
1161:         # --- Original Simple Rule Logic ---
1162:         missing = value is None
1163:         if rule.get("required") and missing:
1164:             errors.append(f"Missing required field '{field}'")
1165:             continue
1166:         if missing:
1167:             continue
1168:         if rule.get("type", "any") != "any" and not EvidenceValidator._check_type(value, rule["type"]):
1169:             errors.append(f"Field '{field}' has incorrect type (expected {rule['type']}')")
1170:             continue
1171:
1172:         if rule.get("min_length") is not None and EvidenceValidator._has_length(value) and len(value) < rule["min_length"]:
1173:             errors.append(f"Field '{field}' length below min_length {rule['min_length']}")
1174:

```

```

1175:         if rule.get("pattern") and isinstance(value, str) and not re.search(rule["pattern"], value):
1176:             errors.append(f"Field '{field}' does not match pattern")
1177:
1178:     # NEW: Process failure_contract from signal pack
1179:     if failure_contract and errors:
1180:         abort_conditions = failure_contract.get("abort_if", [])
1181:         emit_code = failure_contract.get("emit_code", "SIGNAL_ABORT")
1182:         severity = failure_contract.get("severity", "ERROR")
1183:
1184:         for condition in abort_conditions:
1185:             condition_triggered = False
1186:             if condition == "missing_required_element":
1187:                 condition_triggered = any("missing required" in e.lower() for e in errors)
1188:             elif condition == "type_mismatch":
1189:                 condition_triggered = any("incorrect type" in e.lower() for e in errors)
1190:             elif condition == "pattern_mismatch":
1191:                 condition_triggered = any("does not match pattern" in e.lower() for e in errors)
1192:             elif condition == "any_error":
1193:                 condition_triggered = len(errors) > 0
1194:
1195:             if condition_triggered:
1196:                 abort_code = emit_code
1197:                 if severity == "CRITICAL":
1198:                     raise ValueError(
1199:                         f"ABORT[{emit_code}]: Failure contract triggered by condition '{condition}'. Errors: {'; '.join(errors)}"
1200:                     )
1201:                 break
1202:
1203:     valid = not errors
1204:     if errors and na_policy == "abort_on_critical" and not abort_code:
1205:         raise ValueError(f"Evidence validation failed with critical errors: {'; '.join(errors)}")
1206:
1207:     return {
1208:         "valid": valid,
1209:         "errors": errors,
1210:         "warnings": warnings,
1211:         "abort_code": abort_code, # NEW: Track signal-based abort code
1212:         "failure_contract_triggered": abort_code is not None,
1213:     }
1214:
1215:     @staticmethod
1216:     def _resolve(path: str, evidence: dict[str, Any]) -> Any:
1217:         if not path:
1218:             return None
1219:         parts = path.split(".")
1220:         current: Any = evidence
1221:         for part in parts:
1222:             if isinstance(current, dict) and part in current:
1223:                 current = current[part]
1224:             else:
1225:                 return None
1226:         return current
1227:
1228:     @staticmethod
1229:     def _check_type(value: Any, expected: str) -> bool:
1230:         mapping = {

```

```
1231:         "array": (list, tuple),
1232:         "integer": (int,),
1233:         "float": (float, int),
1234:         "string": (str,),
1235:         "boolean": (bool,),
1236:         "object": (dict,),
1237:         "any": (object,),
1238:     }
1239:     return isinstance(value, mapping.get(expected, (object,)))
1240:
1241:     @staticmethod
1242:     def _has_length(value: Any) -> bool:
1243:         return hasattr(value, "__len__")
1244:
1245:     @staticmethod
1246:     def _is_number(value: Any) -> bool:
1247:         try:
1248:             float(value)
1249:             return not isinstance(value, bool)
1250:         except (TypeError, ValueError):
1251:             return False
1252:
1253:
1254:
1255: =====
1256: FILE: src/farfan_pipeline/core/orchestrator/executor_config.py
1257: =====
1258:
1259: from __future__ import annotations
1260:
1261: from dataclasses import dataclass
1262: from typing import Any
1263:
1264:
1265: @dataclass
1266: class ExecutorConfig:
1267:     """
1268:         Lightweight configuration for executors.
1269:
1270:         This is intentionally minimal and only covers the parameters currently
1271:         referenced by wiring/bootstrap code. Extend cautiously if new executor
1272:         settings are required.
1273:     """
1274:
1275:     max_tokens: int | None = None
1276:     temperature: float | None = None
1277:     timeout_s: float | None = None
1278:     retry: int | None = None
1279:     seed: int | None = None
1280:     extra: dict[str, Any] | None = None
1281:
1282:     def __post_init__(self) -> None:
1283:         # Basic type guards without altering semantics
1284:         if self.max_tokens is not None and self.max_tokens <= 0:
1285:             raise ValueError("max_tokens must be positive when provided")
1286:         if self.retry is not None and self.retry < 0:
```

```
1287:             raise ValueError("retry must be non-negative when provided")
1288:
1289:
1290: __all__ = ["ExecutorConfig"]
1291:
1292:
1293:
1294: =====
1295: FILE: src/farfan_pipeline/core/orchestrator/executor_profiler.py
1296: =====
1297:
1298: """Executor performance profiling framework with regression detection and dispensary analytics.
1299:
1300: This module provides comprehensive profiling for executor performance including:
1301: - Per-executor timing, memory, and serialization metrics
1302: - Method call tracking with granular statistics
1303: - Baseline comparison for regression detection
1304: - Performance report generation identifying bottlenecks
1305: - Integration with BaseExecutor for automatic capture
1306: - **METHOD DISPENSARY PATTERN AWARENESS** for tracking monolith reuse
1307:
1308: Architecture:
1309: - ExecutorMetrics: Per-executor performance data
1310: - MethodCallMetrics: Per-method call statistics
1311: - ExecutorProfiler: Main profiler with baseline management + dispensary analytics
1312: - ProfilerContext: Context manager for automatic profiling
1313: - PerformanceReport: Structured report with bottleneck analysis
1314:
1315: METHOD DISPENSARY INTEGRATION:
1316: =====
1317: This profiler is aware of the factory's method dispensary pattern where:
1318: - 30 executors orchestrate methods from ~20 monolith classes
1319: - Methods are called via MethodExecutor.execute(class_name, method_name, **payload)
1320: - Same methods are PARTIALLY reused across different executors
1321: - Dispensary classes: PDET Municipal Plan Analyzer (52+ methods), CausalExtractor (28), etc.
1322:
1323: The profiler tracks:
1324: 1. Which dispensary classes are used by each executor
1325: 2. Method reuse patterns across executors
1326: 3. Performance hotspots within dispensary classes
1327: 4. Executor-specific vs dispensary-wide bottlenecks
1328:
1329: Usage:
1330:     # Basic profiling
1331:     profiler = ExecutorProfiler()
1332:     with profiler.profile_executor("D1-Q1"):
1333:         result = executor.execute(context)
1334:     report = profiler.generate_report()
1335:
1336:     # With dispensary analytics
1337:     dispensary_stats = profiler.get_dispensary_usage_stats()
1338:     # Shows: PDET Municipal Plan Analyzer used by 15 executors, avg 245ms/call
1339: """
1340:
1341: from __future__ import annotations
1342:
```

```
1343: import gc
1344: import json
1345: import logging
1346: import pickle
1347: import time
1348: from collections import defaultdict
1349: from dataclasses import asdict, dataclass, field
1350: from datetime import datetime, timezone
1351: from pathlib import Path
1352: from typing import Any
1353:
1354: logger = logging.getLogger(__name__)
1355:
1356: # Performance thresholds (loaded from canonical_method_catalogue_v2.json via calibration system)
1357: # These are DEFAULT values only - actual thresholds come from method_parameters.json
1358: DEFAULT_HIGH_EXECUTION_TIME_MS = 1000
1359: DEFAULT_HIGH_MEMORY_MB = 100
1360: DEFAULT_HIGH_SERIALIZATION_MS = 100
1361:
1362: # Known dispensary classes from the method dispensary pattern
1363: KNOWN_DISPENSARY_CLASSES = {
1364:     "PDET MunicipalPlanAnalyzer",
1365:     "IndustrialPolicyProcessor",
1366:     "CausalExtractor",
1367:     "FinancialAuditor",
1368:     "BayesianMechanismInference",
1369:     "BayesianCounterfactualAuditor",
1370:     "TextMiningEngine",
1371:     "SemanticAnalyzer",
1372:     "PerformanceAnalyzer",
1373:     "PolicyContradictionDetector",
1374:     "BayesianNumericalAnalyzer",
1375:     "TemporalLogicVerifier",
1376:     "OperationalizationAuditor",
1377:     "PolicyAnalysisEmbedder",
1378:     "SemanticProcessor",
1379:     "AdvancedDAGValidator",
1380:     "TeoriaCambio",
1381:     "ReportingEngine",
1382:     "HierarchicalGenerativeModel",
1383:     "AdaptivePriorCalculator",
1384:     "PolicyTextProcessor",
1385:     "MechanismPartExtractor",
1386:     "CausalInferenceSetup",
1387:     "BeachEvidentialTest",
1388:     "BayesFactorTable",
1389:     "ConfigLoader",
1390:     "CDAFFramework",
1391:     "IndustrialGradeValidator",
1392:     "BayesianConfidenceCalculator",
1393:     "PDFProcessor",
1394: }
1395:
1396:
1397: @dataclass
1398: class MethodCallMetrics:
```

```
1399:     """Metrics for a single method call within an executor.
1400:
1401:     Enhanced to track dispensary pattern usage.
1402:     """
1403:
1404:     class_name: str
1405:     method_name: str
1406:     execution_time_ms: float
1407:     memory_delta_mb: float
1408:     call_count: int = 1
1409:     success: bool = True
1410:     error: str | None = None
1411:     timestamp: str = field(
1412:         default_factory=lambda: datetime.now(timezone.utc).isoformat()
1413:     )
1414:
1415:     @property
1416:     def is_dispensary_method(self) -> bool:
1417:         """Check if this method comes from a known dispensary class."""
1418:         return self.class_name in KNOWN_DISPENSARY_CLASSES
1419:
1420:     @property
1421:     def full_method_name(self) -> str:
1422:         """Get full method name as class.method."""
1423:         return f"{self.class_name}.{self.method_name}"
1424:
1425:     def to_dict(self) -> dict[str, Any]:
1426:         """Convert to dictionary for serialization."""
1427:         data = asdict(self)
1428:         data["is_dispensary_method"] = self.is_dispensary_method
1429:         data["full_method_name"] = self.full_method_name
1430:         return data
1431:
1432:
1433: @dataclass
1434: class ExecutorMetrics:
1435:     """Comprehensive metrics for a single executor execution.
1436:
1437:     Enhanced with dispensary usage tracking.
1438:     """
1439:
1440:     executor_id: str
1441:     execution_time_ms: float
1442:     memory_footprint_mb: float
1443:     memory_peak_mb: float
1444:     serialization_time_ms: float
1445:     serialization_size_bytes: int
1446:     method_calls: list[MethodCallMetrics] = field(default_factory=list)
1447:     call_count: int = 1
1448:     success: bool = True
1449:     error: str | None = None
1450:     timestamp: str = field(
1451:         default_factory=lambda: datetime.now(timezone.utc).isoformat()
1452:     )
1453:     metadata: dict[str, Any] = field(default_factory=dict)
1454:
```

```
1455:     @property
1456:     def total_method_calls(self) -> int:
1457:         """Total number of method calls during execution."""
1458:         return sum(m.call_count for m in self.method_calls)
1459:
1460:     @property
1461:     def dispensary_method_calls(self) -> int:
1462:         """Number of calls to dispensary methods."""
1463:         return sum(m.call_count for m in self.method_calls if m.is_dispensary_method)
1464:
1465:     @property
1466:     def dispensary_usage_ratio(self) -> float:
1467:         """Ratio of dispensary calls to total calls."""
1468:         total = self.total_method_calls
1469:         return self.dispensary_method_calls / total if total > 0 else 0.0
1470:
1471:     @property
1472:     def unique_dispensaries_used(self) -> set[str]:
1473:         """Set of unique dispensary classes used."""
1474:         return {m.class_name for m in self.method_calls if m.is_dispensary_method}
1475:
1476:     @property
1477:     def average_method_time_ms(self) -> float:
1478:         """Average method execution time."""
1479:         if not self.method_calls:
1480:             return 0.0
1481:         return sum(m.execution_time_ms for m in self.method_calls) / len(
1482:             self.method_calls
1483:         )
1484:
1485:     @property
1486:     def slowest_method(self) -> MethodCallMetrics | None:
1487:         """Identify the slowest method call."""
1488:         if not self.method_calls:
1489:             return None
1490:         return max(self.method_calls, key=lambda m: m.execution_time_ms)
1491:
1492:     @property
1493:     def memory_intensive_method(self) -> MethodCallMetrics | None:
1494:         """Identify the most memory-intensive method call."""
1495:         if not self.method_calls:
1496:             return None
1497:         return max(self.method_calls, key=lambda m: abs(m.memory_delta_mb))
1498:
1499:     def to_dict(self) -> dict[str, Any]:
1500:         """Convert to dictionary for serialization."""
1501:         data = asdict(self)
1502:         data["method_calls"] = [m.to_dict() for m in self.method_calls]
1503:         data["total_method_calls"] = self.total_method_calls
1504:         data["dispensary_method_calls"] = self.dispensary_method_calls
1505:         data["dispensary_usage_ratio"] = self.dispensary_usage_ratio
1506:         data["unique_dispensaries_used"] = list(self.unique_dispensaries_used)
1507:         data["average_method_time_ms"] = self.average_method_time_ms
1508:         slowest = self.slowest_method
1509:         data["slowest_method"] = (
1510:             f"{slowest.class_name}.{slowest.method_name}" if slowest else None
```

```
1511:         )
1512:         memory_intensive = self.memory_intensive_method
1513:         data["memory_intensive_method"] = (
1514:             f'{memory_intensive.class_name}.{memory_intensive.method_name}'
1515:             if memory_intensive
1516:             else None
1517:         )
1518:     return data
1519:
1520:
1521: @dataclass
1522: class PerformanceRegression:
1523:     """Detected performance regression for an executor."""
1524:
1525:     executor_id: str
1526:     metric_name: str
1527:     baseline_value: float
1528:     current_value: float
1529:     delta_percent: float
1530:     severity: str
1531:     threshold_exceeded: bool
1532:     recommendation: str
1533:
1534:     def to_dict(self) -> dict[str, Any]:
1535:         """Convert to dictionary for serialization."""
1536:         return asdict(self)
1537:
1538:
1539: @dataclass
1540: class PerformanceReport:
1541:     """Comprehensive performance report with bottleneck analysis.
1542:
1543:     Enhanced with dispensary pattern analytics.
1544:     """
1545:
1546:     timestamp: str
1547:     total_executors: int
1548:     total_execution_time_ms: float
1549:     total_memory_mb: float
1550:     regressions: list[PerformanceRegression] = field(default_factory=list)
1551:     bottlenecks: list[dict[str, Any]] = field(default_factory=list)
1552:     summary: dict[str, Any] = field(default_factory=dict)
1553:     executor_rankings: dict[str, list[str]] = field(default_factory=dict)
1554:     dispensary_analytics: dict[str, Any] = field(default_factory=dict)
1555:
1556:     def to_dict(self) -> dict[str, Any]:
1557:         """Convert to dictionary for serialization."""
1558:         data = asdict(self)
1559:         data["regressions"] = [r.to_dict() for r in self.regressions]
1560:         return data
1561:
1562:
1563: class ExecutorProfiler:
1564:     """Performance profiler with baseline management and regression detection.
1565:
1566:     Tracks per-executor metrics including timing, memory, serialization overhead,
```

```
1567:     and method call counts. Supports baseline comparison for regression detection
1568:     and generates comprehensive performance reports.
1569:
1570:     ENHANCED: Tracks method dispensary pattern usage for monolith reuse analysis.
1571:     """
1572:
1573:     def __init__(
1574:         self,
1575:             baseline_path: Path | str | None = None,
1576:             auto_save_baseline: bool = False,
1577:             memory_tracking: bool = True,
1578:             track_dispensary_usage: bool = True,
1579:             performance_thresholds: dict[str, float] | None = None,
1580:         ) -> None:
1581:             """Initialize the profiler.
1582:
1583:             Args:
1584:                 baseline_path: Path to baseline metrics file (JSON)
1585:                 auto_save_baseline: Automatically update baseline after each run
1586:                 memory_tracking: Enable memory tracking (adds overhead)
1587:                 track_dispensary_usage: Track dispensary class usage patterns
1588:                 performance_thresholds: Performance thresholds from canonical config
1589:                     (execution_time_ms, memory_mb, serialization_ms)
1590:                         If None, uses defaults from method_parameters.json
1591:             """
1592:             self.baseline_path = Path(baseline_path) if baseline_path else None
1593:             self.auto_save_baseline = auto_save_baseline
1594:             self.memory_tracking = memory_tracking
1595:             self.track_dispensary_usage = track_dispensary_usage
1596:
1597:             # Load thresholds from canonical config or use defaults
1598:             self.thresholds = performance_thresholds or self._load_default_thresholds()
1599:
1600:             self.metrics: dict[str, list[ExecutorMetrics]] = defaultdict(list)
1601:             self.baseline_metrics: dict[str, ExecutorMetrics] = {}
1602:             self.regressions: list[PerformanceRegression] = []
1603:
1604:             # Dispensary usage tracking
1605:             self.dispensary_call_counts: dict[str, int] = defaultdict(int)
1606:             self.dispensary_execution_times: dict[str, list[float]] = defaultdict(list)
1607:             self.executor_dispensary_usage: dict[str, set[str]] = defaultdict(set)
1608:
1609:     def _load_default_thresholds(self) -> dict[str, float]:
1610:         """Load default thresholds (can be overridden by canonical config)."""
1611:         return {
1612:             "execution_time_ms": DEFAULT_HIGH_EXECUTION_TIME_MS,
1613:             "memory_mb": DEFAULT_HIGH_MEMORY_MB,
1614:             "serialization_ms": DEFAULT_HIGH_SERIALIZATION_MS,
1615:         }
1616:
1617:         self._psutil = None
1618:         self._psutil_process = None
1619:         if memory_tracking:
1620:             try:
1621:                 import psutil
1622:
```

```
1623:             self._psutil = psutil
1624:             self._psutil_process = psutil.Process()
1625:         except ImportError:
1626:             logger.warning(
1627:                 "psutil not available, memory tracking disabled. "
1628:                 "Install with: pip install psutil"
1629:             )
1630:             self.memory_tracking = False
1631:
1632:         if self.baseline_path and self.baseline_path.exists():
1633:             self.load_baseline(self.baseline_path)
1634:
1635:     def _get_memory_usage_mb(self) -> float:
1636:         """Get current memory usage in MB."""
1637:         if not self.memory_tracking or not self._psutil_process:
1638:             return 0.0
1639:         try:
1640:             return self._psutil_process.memory_info().rss / (1024 * 1024)
1641:         except Exception as exc:
1642:             logger.warning(f"Failed to get memory usage: {exc}")
1643:             return 0.0
1644:
1645:     def profile_executor(self, executor_id: str) -> ProfilerContext:
1646:         """Create a profiling context for an executor.
1647:
1648:         Args:
1649:             executor_id: Unique executor identifier (e.g., "D1-Q1")
1650:
1651:         Returns:
1652:             ProfilerContext for use in with statement
1653:
1654:         Example:
1655:             with profiler.profile_executor("D1-Q1") as ctx:
1656:                 result = executor.execute(context)
1657:                 ctx.add_method_call("TextMiner", "extract", 45.2, 2.1)
1658:             """
1659:             return ProfilerContext(self, executor_id)
1660:
1661:     def record_executor_metrics(
1662:         self, executor_id: str, metrics: ExecutorMetrics
1663:     ) -> None:
1664:         """Record metrics for an executor execution.
1665:
1666:         Args:
1667:             executor_id: Unique executor identifier
1668:             metrics: Collected metrics for the execution
1669:         """
1670:         self.metrics[executor_id].append(metrics)
1671:
1672:         # Track dispensary usage
1673:         if self.track_dispensary_usage:
1674:             self._update_dispensary_stats(executor_id, metrics)
1675:
1676:         if self.baseline_path and self.auto_save_baseline:
1677:             self._update_baseline(executor_id, metrics)
1678:
```

```
1679:     def _update_dispensary_stats(
1680:         self, executor_id: str, metrics: ExecutorMetrics
1681:     ) -> None:
1682:         """Update dispensary usage statistics.
1683:
1684:         Args:
1685:             executor_id: Executor identifier
1686:             metrics: Metrics containing method calls
1687:         """
1688:         for method_call in metrics.method_calls:
1689:             if method_call.is_dispensary_method:
1690:                 class_name = method_call.class_name
1691:
1692:                 # Track call counts
1693:                 self.dispensary_call_counts[class_name] += method_call.call_count
1694:
1695:                 # Track execution times
1696:                 self.dispensary_execution_times[class_name].append(
1697:                     method_call.execution_time_ms
1698:                 )
1699:
1700:                 # Track executor\206\222dispensary usage
1701:                 self.executor_dispensary_usage[executor_id].add(class_name)
1702:
1703:     def _update_baseline(self, executor_id: str, metrics: ExecutorMetrics) -> None:
1704:         """Update baseline with new metrics (running average).
1705:
1706:         Args:
1707:             executor_id: Executor identifier
1708:             metrics: New metrics to incorporate
1709:         """
1710:         if executor_id not in self.baseline_metrics:
1711:             self.baseline_metrics[executor_id] = metrics
1712:         else:
1713:             baseline = self.baseline_metrics[executor_id]
1714:             baseline.execution_time_ms = (
1715:                 baseline.execution_time_ms * 0.8 + metrics.execution_time_ms * 0.2
1716:             )
1717:             baseline.memory_footprint_mb = (
1718:                 baseline.memory_footprint_mb * 0.8 + metrics.memory_footprint_mb * 0.2
1719:             )
1720:             baseline.serialization_time_ms = (
1721:                 baseline.serialization_time_ms * 0.8
1722:                 + metrics.serialization_time_ms * 0.2
1723:             )
1724:             baseline.call_count += 1
1725:
1726:     def detect_regressions(
1727:         self,
1728:         thresholds: dict[str, float] | None = None,
1729:     ) -> list[PerformanceRegression]:
1730:         """Detect performance regressions against baseline.
1731:
1732:         Args:
1733:             thresholds: Regression thresholds for each metric
1734:             (default: {"execution_time_ms": 20.0, "memory_footprint_mb": 30.0})
```

```
1735:
1736:     Returns:
1737:         List of detected regressions
1738:     """
1739:     if thresholds is None:
1740:         thresholds = {
1741:             "execution_time_ms": 20.0,
1742:             "memory_footprint_mb": 30.0,
1743:             "serialization_time_ms": 50.0,
1744:         }
1745:
1746:     regressions: list[PerformanceRegression] = []
1747:
1748:     for executor_id, metric_list in self.metrics.items():
1749:         if not metric_list:
1750:             continue
1751:
1752:         if executor_id not in self.baseline_metrics:
1753:             continue
1754:
1755:         baseline = self.baseline_metrics[executor_id]
1756:         current = metric_list[-1]
1757:
1758:         for metric_name, threshold in thresholds.items():
1759:             baseline_val = getattr(baseline, metric_name, 0.0)
1760:             current_val = getattr(current, metric_name, 0.0)
1761:
1762:             if baseline_val == 0:
1763:                 continue
1764:
1765:             delta_percent = ((current_val - baseline_val) / baseline_val) * 100
1766:
1767:             if delta_percent > threshold:
1768:                 severity = (
1769:                     "critical" if delta_percent > threshold * 2 else "warning"
1770:                 )
1771:                 recommendation = self._generate_recommendation(
1772:                     executor_id, metric_name, delta_percent, current
1773:                 )
1774:
1775:                 regression = PerformanceRegression(
1776:                     executor_id=executor_id,
1777:                     metric_name=metric_name,
1778:                     baseline_value=baseline_val,
1779:                     current_value=current_val,
1780:                     delta_percent=delta_percent,
1781:                     severity=severity,
1782:                     threshold_exceeded=True,
1783:                     recommendation=recommendation,
1784:                 )
1785:                 regressions.append(regression)
1786:
1787:     self.regressions = regressions
1788:     return regressions
1789:
1790: def _generate_recommendation(
```

```

1791:         self,
1792:         executor_id: str,
1793:         metric_name: str,
1794:         delta_percent: float,
1795:         metrics: ExecutorMetrics | None = None,
1796:     ) -> str:
1797:         """Generate optimization recommendation for a regression with dispensary awareness."""
1798:         base_recommendations = {
1799:             "execution_time_ms": (
1800:                 f"Executor {executor_id} execution time increased by {delta_percent:.1f}%. "
1801:             ),
1802:             "memory_footprint_mb": (
1803:                 f"Executor {executor_id} memory usage increased by {delta_percent:.1f}%. "
1804:             ),
1805:             "serialization_time_ms": (
1806:                 f"Executor {executor_id} serialization overhead increased by {delta_percent:.1f}%. "
1807:             ),
1808:         }
1809:
1810:         recommendation = base_recommendations.get(
1811:             metric_name,
1812:             f"Performance degradation detected in {metric_name} ({delta_percent:.1f}%)",
1813:         )
1814:
1815:         # Add dispensary-specific suggestions
1816:         if metrics and self.track_dispensary_usage:
1817:             if metric_name == "execution_time_ms" and metrics.slowest_method:
1818:                 slowest = metrics.slowest_method
1819:                 if slowest.is_dispensary_method:
1820:                     shared_count = len(
1821:                         [
1822:                             eid
1823:                             for eid, dispensaries in self.executor_dispensary_usage.items()
1824:                             if slowest.class_name in dispensaries
1825:                         ]
1826:                     )
1827:                     recommendation += (
1828:                         f"Bottleneck in dispensary method {slowest.full_method_name} "
1829:                         f"({slowest.execution_time_ms:.1f}ms). "
1830:                         f"Consider optimizing this method as it's shared across "
1831:                         f"{shared_count} executors."
1832:                     )
1833:                 else:
1834:                     recommendation += f"Review method call sequence or optimize {slowest.full_method_name}.""
1835:             elif metric_name == "memory_footprint_mb":
1836:                 recommendation += "Check for memory leaks, optimize data structures, or implement streaming."
1837:             elif metric_name == "serialization_time_ms":
1838:                 recommendation += "Reduce result payload size or use more efficient serialization format."
1839:
1840:         return recommendation
1841:
1842:     def identify_bottlenecks(self, top_n: int = 10) -> list[dict[str, Any]]:
1843:         """Identify top bottleneck executors requiring optimization.
1844:
1845:         Args:
1846:             top_n: Number of top bottlenecks to return

```

```
1847:  
1848:     Returns:  
1849:         List of bottleneck descriptors with metrics and recommendations  
1850:     """  
1851:     bottlenecks: list[dict[str, Any]] = []  
1852:  
1853:     for executor_id, metric_list in self.metrics.items():  
1854:         if not metric_list:  
1855:             continue  
1856:  
1857:         avg_metrics = self._compute_average_metrics(metric_list)  
1858:  
1859:         bottleneck_score = (  
1860:             avg_metrics["execution_time_ms"] * 0.5  
1861:             + avg_metrics["memory_footprint_mb"] * 0.3  
1862:             + avg_metrics["serialization_time_ms"] * 0.2  
1863:         )  
1864:  
1865:         bottleneck = {  
1866:             "executor_id": executor_id,  
1867:             "bottleneck_score": bottleneck_score,  
1868:             "avg_execution_time_ms": avg_metrics["execution_time_ms"],  
1869:             "avg_memory_mb": avg_metrics["memory_footprint_mb"],  
1870:             "avg_serialization_ms": avg_metrics["serialization_time_ms"],  
1871:             "total_method_calls": avg_metrics["total_method_calls"],  
1872:             "dispensary_usage_ratio": avg_metrics.get(  
1873:                 "dispensary_usage_ratio", 0.0  
1874:             ),  
1875:             "unique_dispensaries": list(  
1876:                 self.executor_dispensary_usage.get(executor_id, set())  
1877:             ),  
1878:             "slowest_method": avg_metrics["slowest_method"],  
1879:             "memory_intensive_method": avg_metrics["memory_intensive_method"],  
1880:             "recommendation": self._generate_bottleneck_recommendation(  
1881:                 executor_id, avg_metrics  
1882:             ),  
1883:         }  
1884:         bottlenecks.append(bottleneck)  
1885:  
1886:     bottlenecks.sort(key=lambda x: x["bottleneck_score"], reverse=True)  
1887:     return bottlenecks[:top_n]  
1888:  
1889: def _compute_average_metrics(  
1890:     self, metric_list: list[ExecutorMetrics]  
1891: ) -> dict[str, Any]:  
1892:     """Compute average metrics from a list of executor metrics."""  
1893:     if not metric_list:  
1894:         return {}  
1895:  
1896:     return {  
1897:         "execution_time_ms": sum(m.execution_time_ms for m in metric_list)  
1898:         / len(metric_list),  
1899:         "memory_footprint_mb": sum(m.memory_footprint_mb for m in metric_list)  
1900:         / len(metric_list),  
1901:         "serialization_time_ms": sum(m.serialization_time_ms for m in metric_list)  
1902:         / len(metric_list),
```

```

1903:         "total_method_calls": sum(m.total_method_calls for m in metric_list)
1904:         / len(metric_list),
1905:         "dispensary_usage_ratio": sum(m.dispensary_usage_ratio for m in metric_list)
1906:         / len(metric_list),
1907:         "slowest_method": (
1908:             metric_list[-1].slowest_method.class_name
1909:             + "."
1910:             + metric_list[-1].slowest_method.method_name
1911:             if metric_list[-1].slowest_method
1912:             else None
1913:         ),
1914:         "memory_intensive_method": (
1915:             metric_list[-1].memory_intensive_method.class_name
1916:             + "."
1917:             + metric_list[-1].memory_intensive_method.method_name
1918:             if metric_list[-1].memory_intensive_method
1919:             else None
1920:         ),
1921:     }
1922:
1923:     def _generate_bottleneck_recommendation(
1924:         self, _executor_id: str, avg_metrics: dict[str, Any]
1925:     ) -> str:
1926:         """Generate optimization recommendation for a bottleneck with dispensary awareness."""
1927:         recommendations = []
1928:
1929:         if avg_metrics["execution_time_ms"] > self.thresholds["execution_time_ms"]:
1930:             slowest = avg_metrics["slowest_method"]
1931:             if slowest and any(
1932:                 dispensary in slowest for dispensary in KNOWN_DISPENSARY_CLASSES
1933:             ):
1934:                 # Extract class name
1935:                 class_name = slowest.split(".")[0]
1936:                 shared_count = len(
1937:                     [
1938:                         eid
1939:                         for eid, dispensaries in self.executor_dispensary_usage.items()
1940:                         if class_name in dispensaries
1941:                     ]
1942:                 )
1943:                 recommendations.append(
1944:                     f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
1945:                     f"dispensary method {slowest} shared by {shared_count} executors - "
1946:                     f"optimization here benefits multiple executors"
1947:                 )
1948:             else:
1949:                 recommendations.append(
1950:                     f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
1951:                     f"optimize {slowest or 'slow methods'}"
1952:                 )
1953:
1954:         if avg_metrics["memory_footprint_mb"] > self.thresholds["memory_mb"]:
1955:             recommendations.append(
1956:                 f"High memory usage ({avg_metrics['memory_footprint_mb']:.1f}MB): "
1957:                 f"review {avg_metrics['memory_intensive_method']} or 'data structures'"
1958:             )

```

```
1959:
1960:     if avg_metrics["serialization_time_ms"] > self.thresholds["serialization_ms"]:
1961:         recommendations.append(
1962:             f"High serialization overhead ({avg_metrics['serialization_time_ms']:.1f}ms): "
1963:             "reduce payload size"
1964:         )
1965:
1966:     if not recommendations:
1967:         return "Performance acceptable, monitor for regressions"
1968:
1969:     return "; ".join(recommendations)
1970:
1971: def get_dispensary_usage_stats(self) -> dict[str, Any]:
1972:     """Get comprehensive dispensary usage statistics."""
1973:     if not self.track_dispensary_usage:
1974:         return {
1975:             "tracking_enabled": False,
1976:             "message": "Dispensary tracking disabled. Enable with track_dispensary_usage=True",
1977:         }
1978:
1979:     dispensary_stats = {}
1980:
1981:     for dispensary_class in KNOWN_DISPENSARY_CLASSES:
1982:         if dispensary_class not in self.dispensary_call_counts:
1983:             continue
1984:
1985:         call_count = self.dispensary_call_counts[dispensary_class]
1986:         exec_times = self.dispensary_execution_times.get(dispensary_class, [])
1987:
1988:         avg_time = sum(exec_times) / len(exec_times) if exec_times else 0.0
1989:         total_time = sum(exec_times)
1990:
1991:         # Find which executors use this dispensary
1992:         using_executors = [
1993:             eid
1994:             for eid, dispensaries in self.executor_dispensary_usage.items()
1995:                 if dispensary_class in dispensaries
1996:         ]
1997:
1998:         dispensary_stats[dispensary_class] = {
1999:             "total_calls": call_count,
2000:             "avg_execution_time_ms": avg_time,
2001:             "total_execution_time_ms": total_time,
2002:             "used_by_executor_count": len(using_executors),
2003:             "using_executors": using_executors,
2004:             "reuse_factor": call_count / max(len(using_executors), 1),
2005:         }
2006:
2007:         # Sort by total execution time
2008:         sorted_dispensaries = sorted(
2009:             dispensary_stats.items(),
2010:             key=lambda x: x[1]["total_execution_time_ms"],
2011:             reverse=True,
2012:         )
2013:
2014:     return {
```

```
2015:         "tracking_enabled": True,
2016:         "total_dispensaries_used": len(dispensary_stats),
2017:         "total_dispensary_calls": sum(self.dispensary_call_counts.values()),
2018:         "dispensaries": dict(sorted_dispensaries),
2019:         "hottest_dispensaries": [
2020:             {
2021:                 "class": name,
2022:                 "total_time_ms": stats["total_execution_time_ms"],
2023:                 "avg_time_ms": stats["avg_execution_time_ms"],
2024:                 "executor_count": stats["used_by_executor_count"],
2025:                 "reuse_factor": stats["reuse_factor"],
2026:             }
2027:             for name, stats in sorted_dispensaries[:5]
2028:         ],
2029:     },
2030:
2031:     def generate_report(
2032:         self, include_regressions: bool = True, include_bottlenecks: bool = True
2033:     ) -> PerformanceReport:
2034:         """Generate comprehensive performance report.
2035:
2036:         Args:
2037:             include_regressions: Include regression detection
2038:             include_bottlenecks: Include bottleneck analysis
2039:
2040:         Returns:
2041:             PerformanceReport with analysis and recommendations
2042:         """
2043:         regressions = []
2044:         if include_regressions:
2045:             regressions = self.detect_regressions()
2046:
2047:         bottlenecks = []
2048:         if include_bottlenecks:
2049:             bottlenecks = self.identify_bottlenecks()
2050:
2051:         total_execution_time = sum(
2052:             m.execution_time_ms for metrics in self.metrics.values() for m in metrics
2053:         )
2054:         total_memory = sum(
2055:             m.memory_footprint_mb for metrics in self.metrics.values() for m in metrics
2056:         )
2057:
2058:         executor_rankings = {
2059:             "slowest": self._rank_executors_by("execution_time_ms"),
2060:             "memory_intensive": self._rank_executors_by("memory_footprint_mb"),
2061:             "serialization_heavy": self._rank_executors_by("serialization_time_ms"),
2062:         }
2063:
2064:         summary = {
2065:             "total_executors_profiled": len(self.metrics),
2066:             "total_executions": sum(len(m) for m in self.metrics.values()),
2067:             "regressions_detected": len(regressions),
2068:             "critical_regressions": sum(
2069:                 1 for r in regressions if r.severity == "critical"
2070:             ),
2071:             "bottlenecks": bottlenecks,
2072:             "total_time_ms": total_execution_time,
2073:             "avg_time_ms": total_execution_time / len(self.metrics),
2074:             "total_memory_mb": total_memory,
2075:             "avg_memory_mb": total_memory / len(self.metrics),
2076:             "executor_rankings": executor_rankings,
2077:             "regressions": regressions,
2078:         }
2079:
2080:         return PerformanceReport(**summary)
```

```

2071:         "bottlenecks_identified": len(bottlenecks),
2072:         "avg_execution_time_ms": total_execution_time
2073:         / max(1, sum(len(m) for m in self.metrics.values())),
2074:         "avg_memory_mb": total_memory
2075:         / max(1, sum(len(m) for m in self.metrics.values())),
2076:     }
2077:
2078:     # Add dispensary analytics to report
2079:     dispensary_analytics = {}
2080:     if self.track_dispensary_usage:
2081:         dispensary_analytics = self.get_dispensary_usage_stats()
2082:
2083:     return PerformanceReport(
2084:         timestamp=datetime.now(timezone.utc).isoformat(),
2085:         total_executors=len(self.metrics),
2086:         total_execution_time_ms=total_execution_time,
2087:         total_memory_mb=total_memory,
2088:         regressions=regressions,
2089:         bottlenecks=bottlenecks,
2090:         summary=summary,
2091:         executor_rankings=executor_rankings,
2092:         dispensary_analytics=dispensary_analytics,
2093:     )
2094:
2095:     def _rank_executors_by(self, metric_name: str, top_n: int = 10) -> list[str]:
2096:         """Rank executors by a specific metric.
2097:
2098:         Args:
2099:             metric_name: Metric to rank by
2100:             top_n: Number of top executors to return
2101:
2102:         Returns:
2103:             List of executor IDs ranked by metric
2104:         """
2105:         rankings = []
2106:         for executor_id, metric_list in self.metrics.items():
2107:             if not metric_list:
2108:                 continue
2109:             avg_value = sum(getattr(m, metric_name, 0.0) for m in metric_list) / len(
2110:                 metric_list
2111:             )
2112:             rankings.append((executor_id, avg_value))
2113:
2114:         rankings.sort(key=lambda x: x[1], reverse=True)
2115:         return [executor_id for executor_id, _ in rankings[:top_n]]
2116:
2117:     def save_baseline(self, path: Path | str | None = None) -> None:
2118:         """Save current metrics as baseline.
2119:
2120:         Args:
2121:             path: Path to save baseline (uses self.baseline_path if None)
2122:         """
2123:         path = Path(path) if path else self.baseline_path
2124:         if not path:
2125:             raise ValueError("No baseline path specified")
2126:
```

```
2127:         path.parent.mkdir(parents=True, exist_ok=True)
2128:
2129:         baseline_data = {
2130:             executor_id: metrics.to_dict()
2131:             for executor_id, metrics in self.baseline_metrics.items()
2132:         }
2133:
2134:         with open(path, "w", encoding="utf-8") as f:
2135:             json.dump(baseline_data, f, indent=2)
2136:
2137:         logger.info(f"Baseline saved to {path}")
2138:
2139:     def load_baseline(self, path: Path | str) -> None:
2140:         """Load baseline metrics from file.
2141:
2142:         Args:
2143:             path: Path to baseline file
2144:         """
2145:         path = Path(path)
2146:         if not path.exists():
2147:             logger.warning(f"Baseline file not found: {path}")
2148:             return
2149:
2150:         with open(path, encoding="utf-8") as f:
2151:             baseline_data = json.load(f)
2152:
2153:             for executor_id, data in baseline_data.items():
2154:                 method_calls = [
2155:                     MethodCallMetrics(**m) for m in data.pop("method_calls", [])
2156:                 ]
2157:                 # Remove computed properties before reconstructing
2158:                 data.pop("total_method_calls", None)
2159:                 data.pop("dispensary_method_calls", None)
2160:                 data.pop("dispensary_usage_ratio", None)
2161:                 data.pop("unique_dispensaries_used", None)
2162:                 data.pop("average_method_time_ms", None)
2163:                 data.pop("slowest_method", None)
2164:                 data.pop("memory_intensive_method", None)
2165:
2166:                 metrics = ExecutorMetrics(**data, method_calls=method_calls)
2167:                 self.baseline_metrics[executor_id] = metrics
2168:
2169:             logger.info(
2170:                 f"Baseline loaded from {path}: {len(self.baseline_metrics)} executors"
2171:             )
2172:
2173:     def export_report(
2174:         self, report: PerformanceReport, path: Path | str, format: str = "json"
2175:     ) -> None:
2176:         """Export performance report to file.
2177:
2178:         Args:
2179:             report: Performance report to export
2180:             path: Output path
2181:             format: Output format ("json", "markdown", or "html")
2182:         """
2183:
```

```
2183:     path = Path(path)
2184:     path.parent.mkdir(parents=True, exist_ok=True)
2185:
2186:     if format == "json":
2187:         with open(path, "w", encoding="utf-8") as f:
2188:             json.dump(report.to_dict(), f, indent=2)
2189:
2190:     elif format == "markdown":
2191:         self._export_markdown(report, path)
2192:
2193:     elif format == "html":
2194:         self._export_html(report, path)
2195:
2196:     else:
2197:         raise ValueError(f"Unsupported format: {format}")
2198:
2199:     logger.info(f"Report exported to {path}")
2200:
2201: def _export_markdown(self, report: PerformanceReport, path: Path) -> None:
2202:     """Export report as Markdown."""
2203:     lines = [
2204:         "# Executor Performance Report",
2205:         f"**Generated:** {report.timestamp}",
2206:         "",
2207:         "## Summary",
2208:         f"- **Total Executors:** {report.total_executors}",
2209:         f"- **Total Execution Time:** {report.total_execution_time_ms:.2f}ms",
2210:         f"- **Total Memory:** {report.total_memory_mb:.2f}MB",
2211:         f"- **Regressions Detected:** {report.summary.get('regressions_detected', 0)}",
2212:         f"- **Bottlenecks Identified:** {report.summary.get('bottlenecks_identified', 0)}",
2213:         "",
2214:     ]
2215:
2216:     # Dispensary analytics section
2217:     if report.dispensary_analytics.get("tracking_enabled"):
2218:         lines.extend(
2219:             [
2220:                 "## Dispensary Usage Analytics",
2221:                 "",
2222:                 f"- **Total Dispensaries Used:** {report.dispensary_analytics.get('total_dispensaries_used', 0)}",
2223:                 f"- **Total Dispensary Calls:** {report.dispensary_analytics.get('total_dispensary_calls', 0)}",
2224:                 "",
2225:                 "## Hottest Dispensaries",
2226:                 "",
2227:                 " | Rank | Class | Total Time (ms) | Avg Time (ms) | Executors | Reuse Factor |",
2228:                 " |-----|-----|-----|-----|-----|-----|-----|",
2229:             ]
2230:         )
2231:
2232:         for i, disp in enumerate(
2233:             report.dispensary_analytics.get("hottest_dispensaries", []), 1
2234:         ):
2235:             lines.append(
2236:                 f" | {i} | {disp['class']} | {disp['total_time_ms']:.1f} | "
2237:                 f"{disp['avg_time_ms']:.1f} | {disp['executor_count']} | "
2238:                 f"{disp['reuse_factor']:.1f} |"
```

```

2239:         )
2240:         lines.append("")
2241:
2242:     if report.regressions:
2243:         lines.extend(
2244:             [
2245:                 "## Performance Regressions",
2246:                 "",
2247:                 "| Executor | Metric | Baseline | Current | Delta | Severity |",
2248:                 "|-----|-----|-----|-----|-----|-----|",
2249:             ]
2250:         )
2251:     for reg in report.regressions:
2252:         lines.append(
2253:             f"| {reg.executor_id} | {reg.metric_name} | "
2254:             f"{reg.baseline_value:.2f} | {reg.current_value:.2f} | "
2255:             f"{reg.delta_percent:+.1f}% | {reg.severity} |"
2256:         )
2257:     lines.append("")
2258:
2259:     if report.bottlenecks:
2260:         lines.extend(
2261:             [
2262:                 "## Top Bottlenecks",
2263:                 "",
2264:                 "| Rank | Executor | Score | Exec Time | Memory | Dispensaries | Recommendation |",
2265:                 "|-----|-----|-----|-----|-----|-----|-----|",
2266:             ]
2267:         )
2268:     for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
2269:         disp_count = len(bottleneck.get("unique_dispensaries", []))
2270:         lines.append(
2271:             f"| {i} | {bottleneck['executor_id']} | "
2272:             f"{bottleneck['bottleneck_score']:.1f} | "
2273:             f"{bottleneck['avg_execution_time_ms']:.1f}ms | "
2274:             f"{bottleneck['avg_memory_mb']:.1f}MB | "
2275:             f"{disp_count} | "
2276:             f"{bottleneck['recommendation'][:60]}... |"
2277:         )
2278:     lines.append("")
2279:
2280:     with open(path, "w", encoding="utf-8") as f:
2281:         f.write("\n".join(lines))
2282:
2283:     def _export_html(self, report: PerformanceReport, path: Path) -> None:
2284:         """Export report as HTML."""
2285:         html = f"""<!DOCTYPE html>
2286: <html>
2287: <head>
2288:     <title>Executor Performance Report</title>
2289:     <style>
2290:         body {{ font-family: Arial, sans-serif; margin: 20px; }}
2291:         h1, h2 {{ color: #333; }}
2292:         table {{ border-collapse: collapse; width: 100%; margin: 20px 0; }}
2293:         th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
2294:         th {{ background-color: #4CAF50; color: white; }}
```

```
2295:         .critical {{ color: red; font-weight: bold; }}
2296:         .warning {{ color: orange; font-weight: bold; }}
2297:     </style>
2298: </head>
2299: <body>
2300:     <h1>Executor Performance Report</h1>
2301:     <p><strong>Generated:</strong> {report.timestamp}</p>
2302:
2303:     <h2>Summary</h2>
2304:     <ul>
2305:         <li><strong>Total Executors:</strong> {report.total_executors}</li>
2306:         <li><strong>Total Execution Time:</strong> {report.total_execution_time_ms:.2f}ms</li>
2307:         <li><strong>Total Memory:</strong> {report.total_memory_mb:.2f}MB</li>
2308:         <li><strong>Regressions Detected:</strong> {report.summary.get('regressions_detected', 0)}</li>
2309:         <li><strong>Bottlenecks Identified:</strong> {report.summary.get('bottlenecks_identified', 0)}</li>
2310:     </ul>
2311: """
2312:
2313:     # Add dispensary analytics
2314:     if report.dispensary_analytics.get("tracking_enabled"):
2315:         html += """
2316:     <h2>Dispensary Usage Analytics</h2>
2317:     <table>
2318:         <tr>
2319:             <th>Rank</th>
2320:             <th>Dispensary Class</th>
2321:             <th>Total Time (ms)</th>
2322:             <th>Avg Time (ms)</th>
2323:             <th>Executor Count</th>
2324:             <th>Reuse Factor</th>
2325:         </tr>
2326: """
2327:
2328:     for i, disp in enumerate(
2329:         report.dispensary_analytics.get("hottest_dispensaries", []), 1
2330:     ):
2331:         html += f"""
2332:         <tr>
2333:             <td>i</td>
2334:             <td>{disp['class']}</td>
2335:             <td>{disp['total_time_ms']:.1f}</td>
2336:             <td>{disp['avg_time_ms']:.1f}</td>
2337:             <td>{disp['executor_count']}</td>
2338:             <td>{disp['reuse_factor']:.1f}</td>
2339:         </tr>
2340: """
2341:         html += "    </table>\n"
2342:
2343:     if report.regressions:
2344:         html += """
2345:     <h2>Performance Regressions</h2>
2346:     <table>
2347:         <tr>
2348:             <th>Executor</th>
2349:             <th>Metric</th>
2350:             <th>Baseline</th>
```

```
2351:             <th>Current</th>
2352:             <th>Delta</th>
2353:             <th>Severity</th>
2354:         </tr>
2355:     """
2356:     for reg in report.regressions:
2357:         severity_class = reg.severity
2358:         html += f"""
2359:         <tr>
2360:             <td>{reg.executor_id}</td>
2361:             <td>{reg.metric_name}</td>
2362:             <td>{reg.baseline_value:.2f}</td>
2363:             <td>{reg.current_value:.2f}</td>
2364:             <td>{reg.delta_percent:+.1f}</td>
2365:             <td class="{severity_class}">{reg.severity}</td>
2366:         </tr>
2367: """
2368:     html += "    </table>\n"
2369:
2370:     if report.bottlenecks:
2371:         html += """
2372: <h2>Top Bottlenecks</h2>
2373: <table>
2374:     <tr>
2375:         <th>Rank</th>
2376:         <th>Executor</th>
2377:         <th>Score</th>
2378:         <th>Exec Time</th>
2379:         <th>Memory</th>
2380:         <th>Recommendation</th>
2381:     </tr>
2382: """
2383:     for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
2384:         html += f"""
2385:         <tr>
2386:             <td>i</td>
2387:             <td>{bottleneck['executor_id']}
```

```
2407:         self.dispensary_call_counts.clear()
2408:         self.dispensary_execution_times.clear()
2409:         self.executor_dispensary_usage.clear()
2410:
2411:
2412: class ProfilerContext:
2413:     """Context manager for automatic executor profiling.
2414:
2415:     Automatically captures timing, memory, and serialization metrics
2416:     when used with a 'with' statement.
2417:     """
2418:
2419:     def __init__(self, profiler: ExecutorProfiler, executor_id: str) -> None:
2420:         """Initialize profiler context.
2421:
2422:         Args:
2423:             profiler: Parent profiler instance
2424:             executor_id: Executor being profiled
2425:         """
2426:         self.profiler = profiler
2427:         self.executor_id = executor_id
2428:         self.start_time: float = 0.0
2429:         self.start_memory: float = 0.0
2430:         self.method_calls: list[MethodCallMetrics] = []
2431:         self.result: Any = None
2432:         self.error: str | None = None
2433:
2434:     def __enter__(self) -> ProfilerContext:
2435:         """Enter profiling context."""
2436:         self.start_time = time.perf_counter()
2437:         self.start_memory = self.profiler._get_memory_usage_mb()
2438:         gc.collect()
2439:         return self
2440:
2441:     def __exit__(
2442:         self,
2443:         exc_type: type[BaseException] | None,
2444:         exc_val: BaseException | None,
2445:         exc_tb: object,
2446:     ) -> None:
2447:         """Exit profiling context and record metrics."""
2448:         execution_time = (time.perf_counter() - self.start_time) * 1000
2449:         end_memory = self.profiler._get_memory_usage_mb()
2450:         memory_footprint = end_memory - self.start_memory
2451:         memory_peak = max(end_memory, self.start_memory)
2452:
2453:         serialization_time, serialization_size = self._measure_serialization()
2454:
2455:         metrics = ExecutorMetrics(
2456:             executor_id=self.executor_id,
2457:             execution_time_ms=execution_time,
2458:             memory_footprint_mb=memory_footprint,
2459:             memory_peak_mb=memory_peak,
2460:             serialization_time_ms=serialization_time,
2461:             serialization_size_bytes=serialization_size,
2462:             method_calls=self.method_calls,
```

```
2463:             success=exc_type is None,
2464:             error=str(exc_val) if exc_val else None,
2465:         )
2466:
2467:         self.profiler.record_executor_metrics(self.executor_id, metrics)
2468:
2469:     def _measure_serialization(self) -> tuple[float, int]:
2470:         """Measure serialization overhead for the result.
2471:
2472:         Returns:
2473:             Tuple of (serialization_time_ms, serialization_size_bytes)
2474:         """
2475:
2476:         if self.result is None:
2477:             return 0.0, 0
2478:
2479:         try:
2480:             start = time.perf_counter()
2481:             serialized = pickle.dumps(self.result, protocol=pickle.HIGHEST_PROTOCOL)
2482:             serialization_time = (time.perf_counter() - start) * 1000
2483:             serialization_size = len(serialized)
2484:             return serialization_time, serialization_size
2485:         except Exception as exc:
2486:             logger.warning(f"Failed to measure serialization: {exc}")
2487:             return 0.0, 0
2488:
2489:     def add_method_call(
2490:         self,
2491:         class_name: str,
2492:         method_name: str,
2493:         execution_time_ms: float,
2494:         memory_delta_mb: float = 0.0,
2495:         success: bool = True,
2496:         error: str | None = None,
2497:     ) -> None:
2498:         """Add a method call to the profiling context.
2499:
2500:             Args:
2501:                 class_name: Class of the method
2502:                 method_name: Name of the method
2503:                 execution_time_ms: Execution time in milliseconds
2504:                 memory_delta_mb: Memory delta in MB
2505:                 success: Whether the call succeeded
2506:                 error: Error message if failed
2507:
2508:             metrics = MethodCallMetrics(
2509:                 class_name=class_name,
2510:                 method_name=method_name,
2511:                 execution_time_ms=execution_time_ms,
2512:                 memory_delta_mb=memory_delta_mb,
2513:                 success=success,
2514:                 error=error,
2515:             )
2516:             self.method_calls.append(metrics)
2517:
2518:     def set_result(self, result: object) -> None:
2519:         """Set the result for serialization measurement.
```

```
2519:  
2520:     Args:  
2521:         result: Execution result (can be any serializable object)  
2522:         """  
2523:         self.result = result  
2524:  
2525:  
2526:     __all__ = [  
2527:         "ExecutorProfiler",  
2528:         "ProfilerContext",  
2529:         "ExecutorMetrics",  
2530:         "MethodCallMetrics",  
2531:         "PerformanceRegression",  
2532:         "PerformanceReport",  
2533:         "KNOWN_DISPENSARY_CLASSES",  
2534:     ]  
2535:  
2536:  
2537:  
2538: =====  
2539: FILE: src/farfan_pipeline/core/orchestrator/executors.py  
2540: =====  
2541:  
2542: """  
2543: executors.py - Phase 2: Executor Orchestration for Policy Document Analysis  
2544:  
2545: This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate  
2546: methods from the core module to extract raw evidence from Colombian municipal  
2547: development plans (PDET/PDM documents).  
2548:  
2549: Architecture:  
2550: - Each executor is independent and receives a canonical context package  
2551: - Methods execute in configured order; any failure causes executor failure  
2552: - Outputs are Python dicts/lists matching JSON contract specifications  
2553: - Executors are injected via MethodExecutor factory pattern  
2554:  
2555: Usage:  
2556:     from factory import run_executor  
2557:     result = run_executor("D1-Q1", context_package)  
2558: """  
2559:  
2560: from __future__ import annotations  
2561:  
2562: import sys  
2563: import logging  
2564: from typing import Dict, List, Any, Optional  
2565: from abc import ABC, abstractmethod  
2566: from dataclasses import dataclass  
2567:  
2568: from farfan_pipeline.core.canonical_notation import CanonicalDimension, get_dimension_info  
2569: from farfan_pipeline.core.orchestrator.core import MethodExecutor  
2570: from farfan_pipeline.core.orchestrator.factory import build_processor  
2571: from farfan_pipeline.core.orchestrator.memory_safety import (  
2572:     MemorySafetyGuard,  
2573:     MemorySafetyConfig,  
2574:     ExecutorType,
```

```

2575:     create_default_guard,
2576: )
2577: from farfan_pipeline.processing.policy_processor import CausalDimension
2578:
2579: logger = logging.getLogger(__name__)
2580:
2581: # Canonical question labels (only defined when verified in repo)
2582: CANONICAL_QUESTION_LABELS = {
2583:     "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
2584:     "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
2585:     "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
2586:     "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
2587:     "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
2588:     "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
2589: }
2590:
2591: # Epistemic taxonomy per method (focused on executors expanded in this iteration)
2592: EPISTEMIC_TAGS = {
2593:     ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
2594:     ("FinancialAuditor", "_match_program_to_node"): ["structural"],
2595:     ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
2596:     ("PDET MunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial", "normative"],
2597:     ("PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial", "statistical"],
2598:     ("PDET MunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
2599:     ("PDET MunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
2600:     ("PDET MunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
2601:     ("PDET MunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
2602:     ("PDET MunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
2603:     ("PDET MunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
2604:     ("PDET MunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic", "structural"],
2605:     ("PDET MunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
2606:     ("PDET MunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical", "causal"],
2607:     ("PDET MunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
2608:     ("PDET MunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
2609:     ("PDET MunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
2610:     ("PDET MunicipalPlanAnalyzer", "_identify_cofounders"): ["causal", "consistency"],
2611:     ("PDET MunicipalPlanAnalyzer", "_effect_to_dict"): [descriptive],
2612:     ("PDET MunicipalPlanAnalyzer", "_scenario_to_dict"): [descriptive],
2613:     ("PDET MunicipalPlanAnalyzer", "_get_spanish_stopwords"): [semantic],
2614:     ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical", "bayesian"],
2615:     ("AdaptivePriorCalculator", "_adjust_domain_weights"): [statistical],
2616:     ("BayesianMechanismInference", "_test_sufficiency"): [statistical, "bayesian"],
2617:     ("BayesianMechanismInference", "_test_necessity"): [statistical, "bayesian"],
2618:     ("BayesianMechanismInference", "_log_refactored_components"): [implementation],
2619:     ("BayesianMechanismInference", "_infer_activity_sequence"): [causal],
2620:     ("BayesianMechanismInference", "infer_mechanisms"): [causal, "bayesian"],
2621:     ("AdvancedDAGValidator", "calculateACYClicity_pvalue"): [statistical, "consistency"],
2622:     ("AdvancedDAGValidator", "_isAcyclic"): [structural, "consistency"],
2623:     ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): [statistical, "bayesian"],
2624:     ("AdvancedDAGValidator", "_calculate_confidence_interval"): [statistical],
2625:     ("AdvancedDAGValidator", "_calculate_statistical_power"): [statistical],
2626:     ("AdvancedDAGValidator", "_generate_subgraph"): [structural],
2627:     ("AdvancedDAGValidator", "_get_node_validator"): [implementation],
2628:     ("AdvancedDAGValidator", "_create_empty_result"): [descriptive],
2629:     ("AdvancedDAGValidator", "_initialize_rng"): [implementation],
2630:     ("AdvancedDAGValidator", "get_graph_stats"): [structural],

```

```
2631: ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
2632: ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
2633: ("AdvancedDAGValidator", "add_node"): ["structural"],
2634: ("AdvancedDAGValidator", "add_edge"): ["structural"],
2635: ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
2636: ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
2637: ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
2638: ("IndustrialGradeValidator", "benchmark_operation"): ["implementation"],
2639: ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
2640: ("IndustrialGradeValidator", "log_metric"): ["implementation"],
2641: ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
2642: ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
2643: ("HierarchicalGenerativeModel", "calculate_ess"): ["statistical"],
2644: ("HierarchicalGenerativeModel", "calculate_likelihood"): ["statistical"],
2645: ("HierarchicalGenerativeModel", "calculate_r_hat"): ["statistical"],
2646: ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
2647: ("ReportingEngine", "calculate_quality_score"): ["normative", "statistical"],
2648: ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
2649: ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
2650: ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency", "statistical"],
2651: ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
2652: ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
2653: ("PolicyAnalysisEmbedder", "apply_mmr"): ["semantic"],
2654: ("PolicyAnalysisEmbedder", "generate_query_from_pdq"): ["semantic"],
2655: ("PolicyAnalysisEmbedder", "filter_by_pdq"): ["semantic"],
2656: ("PolicyAnalysisEmbedder", "extract_numerical_values"): ["statistical"],
2657: ("PolicyAnalysisEmbedder", "compute_overall_confidence"): ["statistical", "normative"],
2658: ("PolicyAnalysisEmbedder", "embed_texts"): ["semantic"],
2659: ("SemanticAnalyzer", "classify_policy_domain"): ["semantic"],
2660: ("SemanticAnalyzer", "empty_semantic_cube"): ["descriptive"],
2661: ("SemanticAnalyzer", "classify_cross_cutting_themes"): ["semantic"],
2662: ("SemanticAnalyzer", "classify_value_chain_link"): ["semantic"],
2663: ("SemanticAnalyzer", "vectorize_segments"): ["semantic"],
2664: ("SemanticAnalyzer", "calculate_semantic_complexity"): ["semantic"],
2665: ("SemanticAnalyzer", "process_segment"): ["semantic"],
2666: ("PDET MunicipalPlanAnalyzer", "entity_to_dict"): ["descriptive"],
2667: ("PDET MunicipalPlanAnalyzer", "quality_to_dict"): ["descriptive", "normative"],
2668: ("PDET MunicipalPlanAnalyzer", "deduplicate_tables"): ["structural", "implementation"],
2669: ("PDET MunicipalPlanAnalyzer", "indicator_to_dict"): ["descriptive"],
2670: ("PDET MunicipalPlanAnalyzer", "generate_recommendations"): ["normative"],
2671: ("PDET MunicipalPlanAnalyzer", "simulate_intervention"): ["causal", "statistical"],
2672: ("PDET MunicipalPlanAnalyzer", "identify_causal_nodes"): ["structural", "causal"],
2673: ("PDET MunicipalPlanAnalyzer", "match_text_to_node"): ["semantic", "structural"],
2674: ("TeoriaCambio", "validar_orden_causal"): ["causal", "consistency"],
2675: ("TeoriaCambio", "generar_sugerencias_internas"): ["normative"],
2676: ("TeoriaCambio", "extraer_categorias"): ["semantic"],
2677: ("BayesianMechanismInference", "extract_observations"): ["semantic", "causal"],
2678: ("BayesianMechanismInference", "generate_necessity_remediation"): ["normative", "causal"],
2679: ("BayesianMechanismInference", "quantify_uncertainty"): ["statistical", "bayesian"],
2680: ("CausalExtractor", "build_type_hierarchy"): ["structural"],
2681: ("CausalExtractor", "check_structuralViolation"): ["structural", "consistency"],
2682: ("CausalExtractor", "calculate_type_transition_prior"): ["statistical", "bayesian"],
2683: ("CausalExtractor", "calculate_textual_proximity"): ["semantic"],
2684: ("CausalExtractor", "calculate_language_specificity"): ["semantic"],
2685: ("CausalExtractor", "calculate_composite_likelihood"): ["statistical", "semantic"],
2686: ("CausalExtractor", "assess_financial_consistency"): ["financial", "consistency"],
```

```

2687:     ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
2688:     ("CausalExtractor", "_extract_goals"): ["semantic"],
2689:     ("CausalExtractor", "_parse_goal_context"): ["semantic"],
2690:     ("CausalExtractor", "_classify_goal_type"): ["semantic"],
2691:     ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
2692:     ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
2693:     ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
2694:     ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
2695:     ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural", "semantic"],
2696:     ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
2697:     ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
2698:     ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
2699:     ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
2700:     ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
2701:     ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
2702:     ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal", "normative"],
2703:     ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal", "consistency"],
2704:     ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
2705:     ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal", "statistical"],
2706:     ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
2707:     ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive", "implementation"],
2708:     ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural", "semantic"],
2709:     ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
2710:     ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
2711:     ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
2712:     ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
2713:     ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
2714:     ("PDET MunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
2715:     ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
2716: }
2717:
2718:
2719: class BaseExecutor(ABC):
2720:     """
2721:         Base class for all executors with standardized execution template.
2722:         All executors must implement execute() and return structured evidence.
2723:
2724:         Includes systematic memory safety guards for processing large objects.
2725:     """
2726:
2727:     def __init__(self, executor_id: str, config: Dict[str, Any], method_executor: MethodExecutor, profiler: Any | None = None):
2728:         self.executor_id = executor_id
2729:         self.config = config
2730:         if not isinstance(method_executor, MethodExecutor):
2731:             raise RuntimeError("A valid MethodExecutor instance is required for executor injection.")
2732:         self.method_executor = method_executor
2733:         self.execution_log = []
2734:         self._dimension_info = None
2735:
2736:         memory_config = config.get("memory_safety", {})
2737:         if isinstance(memory_config, dict):
2738:             self.memory_guard = MemorySafetyGuard(MemorySafetyConfig(**memory_config))
2739:         else:
2740:             self.memory_guard = create_default_guard()
2741:
2742:         self.executor_type = self._determine_executor_type()

```

```
2743:         self._profiler = profiler
2744:         self._profiler_context: Any | None = None
2745:
2746:     def _determine_executor_type(self) -> ExecutorType:
2747:         """Determine memory limit type based on executor ID and primary methods."""
2748:         executor_id_lower = self.executor_id.lower()
2749:
2750:         if "d3-q2" in executor_id_lower or "d3-q3" in executor_id_lower:
2751:             return ExecutorType.ENTITY
2752:         elif "d6-q1" in executor_id_lower or "d6-q2" in executor_id_lower:
2753:             return ExecutorType.DAG
2754:         elif "d5-q" in executor_id_lower or "d6-q3" in executor_id_lower:
2755:             return ExecutorType.CAUSAL_EFFECTS
2756:         elif "d1-q" in executor_id_lower or "d2-q" in executor_id_lower:
2757:             return ExecutorType.SEMANTIC
2758:         elif "d3-q" in executor_id_lower or "d4-q" in executor_id_lower:
2759:             return ExecutorType.FINANCIAL
2760:         else:
2761:             return ExecutorType.GENERIC
2762:
2763:     @property
2764:     def dimension_info(self):
2765:         """Lazy-loaded dimension information to avoid redundant metadata fetches."""
2766:         if self._dimension_info is None:
2767:             try:
2768:                 dim_key = self.executor_id.split("-")[0].replace("D", "D")
2769:                 self._dimension_info = get_dimension_info(dim_key)
2770:             except (KeyError, ValueError, IndexError) as e:
2771:                 logger.warning(f"Failed to load dimension info for {self.executor_id}: {e}")
2772:                 self._dimension_info = None
2773:         return self._dimension_info
2774:
2775:     def _safe_process_object(self, obj: Any, label: str = "object") -> Any:
2776:         """Process object with memory safety guards.
2777:
2778:         Args:
2779:             obj: Object to process (entities, DAG, causal effects, etc.)
2780:             label: Human-readable label for logging
2781:
2782:         Returns:
2783:             Memory-safe processed object
2784:         """
2785:         processed_obj, metrics = self.memory_guard.check_and_process(
2786:             obj, self.executor_type, label
2787:         )
2788:
2789:         if metrics.was_truncated or metrics.was_sampled:
2790:             logger.info(
2791:                 f"{self.executor_id}: Applied {metrics.fallback_strategy} to {label} - "
2792:                 f"size reduced from {metrics.object_size_bytes / (1024*1024):.2f}MB "
2793:                 f"({metrics.elements_before} elements) to "
2794:                 f"({metrics.json_size_bytes / (1024*1024):.2f}MB "
2795:                 f"({metrics.elements_after} elements)"
2796:             )
2797:
2798:         return processed_obj
```

```
2799:  
2800:     def _safe_process_list(self, items: List[Any], label: str = "list") -> List[Any]:  
2801:         """Process list with memory safety guards."""  
2802:         return self._safe_process_object(items, label)  
2803:  
2804:     def _safe_process_dict(self, data: Dict[str, Any], label: str = "dict") -> Dict[str, Any]:  
2805:         """Process dict with memory safety guards."""  
2806:         return self._safe_process_object(data, label)  
2807:  
2808:     def _get_memory_metrics_summary(self) -> Dict[str, Any]:  
2809:         """Get summary of memory operations for this executor."""  
2810:         return self.memory_guard.get_metrics_summary()  
2811:  
2812:     def _validate_context(self, context: Dict[str, Any]) -> None:  
2813:         """  
2814:             Fail fast on malformed contexts.  
2815:  
2816:             Raises:  
2817:                 ValueError: If required context keys are missing  
2818:             """  
2819:             if not isinstance(context, dict):  
2820:                 raise ValueError(f"Context must be a dict, got {type(context).__name__}")  
2821:  
2822:             required = ["document_text"]  
2823:             missing = [k for k in required if k not in context]  
2824:             if missing:  
2825:                 raise ValueError(f"Context missing required keys: {missing}")  
2826:  
2827:     @abstractmethod  
2828:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:  
2829:         """  
2830:             Execute configured methods and return raw evidence.  
2831:  
2832:             Args:  
2833:                 context: Canonical package with document, tables, metadata  
2834:  
2835:             Returns:  
2836:                 Dict with raw_evidence, metadata, execution_metrics  
2837:  
2838:             Raises:  
2839:                 ExecutorFailure: If any method fails  
2840:             """  
2841:             pass  
2842:  
2843:     def _log_method_execution(self, class_name: str, method_name: str,  
2844:                             success: bool, result: Any = None, error: str = None):  
2845:         """Track method execution for debugging and traceability."""  
2846:         self.execution_log.append({  
2847:             "class": class_name,  
2848:             "method": method_name,  
2849:             "success": success,  
2850:             "result_type": type(result).__name__ if result else None,  
2851:             "error": error  
2852:         })  
2853:  
2854:     def _execute_method(self, class_name: str, method_name: str,
```

```
2855:             context: Dict[str, Any], **kwargs) -> Any:
2856:         """
2857:             Execute a single method with error handling and profiling.
2858:
2859:             Raises:
2860:                 ExecutorFailure: If method execution fails
2861:             """
2862:             import time
2863:             start_time = time.perf_counter()
2864:             start_memory = 0.0
2865:
2866:             if self._profiler and self._profiler.memory_tracking:
2867:                 start_memory = self._profiler._get_memory_usage_mb()
2868:
2869:             try:
2870:                 method = self._get_method(class_name, method_name)
2871:                 result = method(context, **kwargs)
2872:
2873:                 execution_time = (time.perf_counter() - start_time) * 1000
2874:                 memory_delta = 0.0
2875:                 if self._profiler and self._profiler.memory_tracking:
2876:                     memory_delta = self._profiler._get_memory_usage_mb() - start_memory
2877:
2878:                 self._log_method_execution(class_name, method_name, True, result)
2879:
2880:                 if self._profiler_context:
2881:                     self._profiler_context.add_method_call(
2882:                         class_name, method_name, execution_time, memory_delta, True
2883:                     )
2884:
2885:             return result
2886:         except Exception as e:
2887:             execution_time = (time.perf_counter() - start_time) * 1000
2888:             memory_delta = 0.0
2889:             if self._profiler and self._profiler.memory_tracking:
2890:                 memory_delta = self._profiler._get_memory_usage_mb() - start_memory
2891:
2892:             self._log_method_execution(class_name, method_name, False, error=str(e))
2893:
2894:             if self._profiler_context:
2895:                 self._profiler_context.add_method_call(
2896:                     class_name, method_name, execution_time, memory_delta, False, str(e)
2897:                 )
2898:
2899:             raise ExecutorFailure(
2900:                 f"Executor {self.executor_id} failed: {class_name}.{method_name} - {str(e)}"
2901:             ) from e
2902:
2903:     def _get_method(self, class_name: str, method_name: str):
2904:         """Retrieve method using MethodExecutor to enforce routed execution."""
2905:         if not isinstance(self.method_executor, MethodExecutor):
2906:             raise RuntimeError(f"Invalid method executor provided: {type(self.method_executor).__name__}")
2907:
2908:     def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
2909:         payload: Dict[str, Any] = {}
2910:         if context:
```

```
2911:         payload.update(context)
2912:     if kwargs:
2913:         payload.update(kwargs)
2914:     return self.method_executor.execute(
2915:         class_name=class_name,
2916:         method_name=method_name,
2917:         **payload,
2918:     )
2919:
2920:     return _wrapped
2921:
2922: def execute_with_profiling(self, context: Dict[str, Any]) -> Dict[str, Any]:
2923:     """Execute with automatic profiling if profiler is attached.
2924:
2925:     Args:
2926:         context: Canonical package with document, tables, metadata
2927:
2928:     Returns:
2929:         Dict with raw_evidence, metadata, execution_metrics
2930:     """
2931:     if self._profiler:
2932:         with self._profiler.profile_executor(self.executor_id) as ctx:
2933:             self._profiler_context = ctx
2934:             try:
2935:                 result = self.execute(context)
2936:                 ctx.set_result(result)
2937:             return result
2938:         finally:
2939:             self._profiler_context = None
2940:     else:
2941:         return self.execute(context)
2942:
2943:
2944: @dataclass
2945: class ExecutorResult:
2946:     """
2947:         Standardized result container for executor execution.
2948:         Ensures type safety and verifiable structure.
2949:     """
2950:     executor_id: str
2951:     success: bool
2952:     data: Optional[Dict[str, Any]]
2953:     error: Optional[str]
2954:     execution_time_ms: int
2955:     memory_usage_mb: float
2956:
2957:
2958: class ExecutorFailure(Exception):
2959:     """Raised when any method in an executor fails."""
2960:     pass
2961:
2962:
2963: # =====
2964: # DIMENSION 1: DIAGNOSTICS & INPUTS
2965: # =====
2966:
```

```
2967: class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
2968:     """
2969:         Extracts numeric data, reference years, and official sources as baseline.
2970:
2971:     Methods (from D1-Q1):
2972:         - TextMiningEngine.diagnose_critical_links
2973:         - TextMiningEngine._analyze_link_text
2974:         - IndustrialPolicyProcessor.process
2975:         - IndustrialPolicyProcessor._match_patterns_in_sentences
2976:         - IndustrialPolicyProcessor._extract_point_evidence
2977:         - CausalExtractor._extract_goals
2978:         - CausalExtractor._parse_goal_context
2979:         - FinancialAuditor._parse_amount
2980:         - PDET MunicipalPlanAnalyzer._extract_financial_amounts
2981:         - PDET MunicipalPlanAnalyzer._extract_from_budget_table
2982:         - PolicyContradictionDetector._extract_quantitative_claims
2983:         - PolicyContradictionDetector._parse_number
2984:         - PolicyContradictionDetector._statistical_significance_test
2985:         - BayesianNumericalAnalyzer.evaluate_policy_metric
2986:         - BayesianNumericalAnalyzer.compare_policies
2987:         - SemanticProcessor.chunk_text
2988:         - SemanticProcessor.embed_single
2989:     """
2990:
2991:     def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str, Any]:
2992:         if context is None:
2993:             context = dict(kwargs)
2994:         raw_evidence: Dict[str, Any] = {}
2995:
2996:         # The new implementation requires manual instantiation of some components
2997:         # because the dependency injection logic was part of the old MethodExecutor.
2998:         # This will be revisited when refactoring the executor base class.
2999:         ontology = self.method_executor.shared_instances.get("MunicipalOntology")
3000:
3001:         # Step 0: Initial processing of the document text
3002:         raw_text = context.get("raw_text", "")
3003:         sentences = self._execute_method("PolicyTextProcessor", "segment_into_sentences", context, text=raw_text)
3004:
3005:         # Step 1: Semantic and Performance Analysis (Prerequisites for TextMiningEngine)
3006:         semantic_cube = self._execute_method("SemanticAnalyzer", "extract_semantic_cube", context, document_segments=sentences)
3007:         performance_analysis = self._execute_method("PerformanceAnalyzer", "analyze_performance", context, semantic_cube=semantic_cube)
3008:
3009:         # Step 2: Identify critical data-bearing sections
3010:         critical_links = self._execute_method(
3011:             "TextMiningEngine", "diagnose_critical_links", context,
3012:             semantic_cube=semantic_cube,
3013:             performance_analysis=performance_analysis
3014:         )
3015:
3016:         # The output of diagnose_critical_links is complex. Let's assume it contains segments for _analyze_link_text
3017:         link_analysis_segments = critical_links.get("critical_links", {}).get(next(iter(critical_links.get("critical_links", {}))), None), {}).get("text_anal
ysis", {}).get("keywords", [])
3018:         link_analysis = self._execute_method(
3019:             "TextMiningEngine", "_analyze_link_text", context,
3020:             segments=[{"text": s} for s in link_analysis_segments] # _analyze_link_text expects list of dicts
3021:     )
```

```
3022:  
3023:     # Step 3: Extract structured quantitative claims from the whole document  
3024:     processed_sections = self._execute_method(  
3025:         "IndustrialPolicyProcessor", "process", context,  
3026:         raw_text=raw_text  
3027:     )  
3028:     # We need compiled patterns. This is a challenge. Let's assume the processor can get them.  
3029:     compiled_patterns = self._execute_method("IndustrialPolicyProcessor", "_compile_pattern_registry", context)  
3030:  
3031:     pattern_matches, _ = self._execute_method(  
3032:         "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,  
3033:         compiled_patterns=compiled_patterns.get(CausalDimension.D1_INSUMOS, {}).get('diagnostico_cuantitativo', []),  
3034:         relevant_sentences=sentences  
3035:     )  
3036:  
3037:     point_evidence_list = []  
3038:     for point_code in self._execute_method("IndustrialPolicyProcessor", "_build_point_patterns", context):  
3039:         point_evidence = self._execute_method(  
3040:             "IndustrialPolicyProcessor", "_extract_point_evidence", context,  
3041:             text=raw_text,  
3042:             sentences=sentences,  
3043:             point_code=point_code  
3044:         )  
3045:         point_evidence_list.append(point_evidence)  
3046:  
3047:     # Step 4: Parse numerical amounts and baseline data  
3048:     all_text = " ".join(pattern_matches)  
3049:     parsed_amounts = self._execute_method( "FinancialAuditor", "_parse_amount", context, value=all_text)  
3050:  
3051:     financial_amounts = self._execute_method(  
3052:         "PDET Municipal Plan Analyzer", "_extract_financial_amounts", context,  
3053:         text=raw_text, tables=context.get("tables", [])  
3054:     )  
3055:  
3056:     # This method needs a dataframe, which is not available in the context.  
3057:     # I will skip this call for now.  
3058:     # budget_table_data = self._execute_method(  
3059:     #     "PDET Municipal Plan Analyzer", "_extract_from_budget_table", context  
3060:     # )  
3061:     budget_table_data = None  
3062:  
3063:     # Step 5: Extract temporal context (reference years)  
3064:     goals = self._execute_method( "CausalExtractor", "_extract_goals", context, text=raw_text)  
3065:  
3066:     goal_contexts = []  
3067:     if isinstance(goals, list):  
3068:         for goal in goals:  
3069:             goal_id = goal.id  
3070:             goal_context_str = goal.text  
3071:             if goal_id and goal_context_str:  
3072:                 res = self._execute_method(  
3073:                     "CausalExtractor", "_parse_goal_context", context,  
3074:                     goal_id=goal_id,  
3075:                     context=goal_context_str  
3076:                 )  
3077:                 if res:
```

```
3078:             goal_contexts.append(res)
3079:
3080:     # Step 6: Validate quantitative claims
3081:     quant_claims = self._execute_method( "PolicyContradictionDetector", "_extract_quantitative_claims", context, text=raw_text)
3082:
3083:     parsed_numbers = []
3084:     if isinstance(quant_claims, list):
3085:         for claim in quant_claims:
3086:             res = self._execute_method(
3087:                 "PolicyContradictionDetector", "_parse_number", context,
3088:                 text=claim.get("raw_text")
3089:             )
3090:             if res is not None:
3091:                 parsed_numbers.append(res)
3092:
3093:     significance_test = None
3094:     if len(parsed_numbers) >= 2:
3095:         # The method expects claims, not just numbers. Let's create dummy claims.
3096:         claim_a = {'value': parsed_numbers[0]}
3097:         claim_b = {'value': parsed_numbers[1]}
3098:         significance_test = self._execute_method(
3099:             "PolicyContradictionDetector", "_statistical_significance_test", context,
3100:             claim_a=claim_a,
3101:             claim_b=claim_b
3102:         )
3103:
3104:     # Step 7: Evaluate baseline quality and compare
3105:     metric_evaluation = self._execute_method(
3106:         "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
3107:         observed_values=parsed_numbers
3108:     )
3109:
3110:     policy_comparison = None
3111:     if metric_evaluation and "posterior_samples" in metric_evaluation:
3112:         policy_comparison = self._execute_method(
3113:             "BayesianNumericalAnalyzer", "compare_policies", context,
3114:             policy_a_values=[s['coherence'] for s in metric_evaluation.get("posterior_samples", [])],
3115:             policy_b_values=context.get("baseline_samples", []) # Assuming baseline samples exist
3116:         )
3117:
3118:     # Step 8: Semantic validation of sources
3119:     text_chunks = self._execute_method( "SemanticProcessor", "chunk_text", context, text=raw_text, preserve_structure=True)
3120:
3121:     embeddings = []
3122:     if isinstance(text_chunks, list):
3123:         texts_to_embed = [chunk['content'] for chunk in text_chunks if 'content' in chunk]
3124:         embeddings = self._execute_method( "SemanticProcessor", "_embed_batch", context, texts=texts_to_embed)
3125:
3126:     # Assemble raw evidence
3127:     raw_evidence = {
3128:         "numeric_data": parsed_numbers,
3129:         "reference_years": [gc.year for gc in goal_contexts if gc and hasattr(gc, 'year')],
3130:         "official_sources": point_evidence_list,
3131:         "financial_baseline": financial_amounts,
3132:         "budget_tables": budget_table_data,
3133:         "significance_results": significance_test,
```

```
3134:         "metric_evaluation": metric_evaluation,
3135:         "policy_comparison": policy_comparison,
3136:         "goal_contexts": [res.text if res else None for res in goal_contexts],
3137:         "quantitative_claims": quant_claims,
3138:         "processed_sections": processed_sections,
3139:         "pattern_matches": pattern_matches,
3140:         "link_analysis": link_analysis,
3141:         "source_embeddings": embeddings
3142:     }
3143:
3144:     return {
3145:         "executor_id": self.executor_id,
3146:         "raw_evidence": raw_evidence,
3147:         "metadata": {
3148:             "methods_executed": [log["method"] for log in self.execution_log],
3149:             "total_numeric_claims": len(parsed_numbers or []),
3150:             "sources_identified": len(point_evidence_list)
3151:         },
3152:         "execution_metrics": {
3153:             "methods_count": len(self.execution_log),
3154:             "all_succeeded": all(log["success"] for log in self.execution_log)
3155:         }
3156:     }
3157:
3158:
3159: class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
3160:     """
3161:     Quantifies problem magnitude, gaps, and identifies data limitations.
3162:
3163:     Methods (from D1-Q2):
3164:     - OperationalizationAuditor._audit_direct_evidence
3165:     - OperationalizationAuditor._audit_systemic_risk
3166:     - FinancialAuditor._detect_allocation_gaps
3167:     - BayesianMechanismInference._detect_gaps
3168:     - PDET MunicipalPlanAnalyzer._generate_optimal_remediations
3169:     - PDET MunicipalPlanAnalyzer._simulate_intervention
3170:     - BayesianCounterfactualAuditor.counterfactual_query
3171:     - BayesianCounterfactualAuditor._test_effect_stability
3172:     - PolicyContradictionDetector._detect_numerical_inconsistencies
3173:     - PolicyContradictionDetector._calculate_numerical_divergence
3174:     - BayesianConfidenceCalculator.calculate_posterior
3175:     - PerformanceAnalyzer.analyze_performance
3176:     """
3177:
3178:     def execute(self, context: Dict[str, Any] | None = None, **kwargs: Any) -> Dict[str, Any]:
3179:         if context is None:
3180:             context = dict(kwargs)
3181:         raw_evidence: Dict[str, Any] = {}
3182:
3183:         # Step 1: Audit evidence completeness
3184:         direct_evidence_audit = self._execute_method(
3185:             "OperationalizationAuditor", "_audit_direct_evidence", context
3186:         )
3187:         systemic_risk_audit = self._execute_method(
3188:             "OperationalizationAuditor", "_audit_systemic_risk", context
3189:         )
```

```
3190:  
3191:      # Step 2: Detect gaps in resource allocation and mechanisms  
3192:      allocation_gaps = self._execute_method(  
3193:          "FinancialAuditor", "_detect_allocation_gaps", context  
3194:      )  
3195:      mechanism_gaps = self._execute_method(  
3196:          "BayesianMechanismInference", "_detect_gaps", context  
3197:      )  
3198:  
3199:      # Step 3: Generate optimal remediations and simulate interventions  
3200:      remediations = self._execute_method(  
3201:          "PDET MunicipalPlanAnalyzer", "_generate_optimal_remediations", context,  
3202:          gaps=allocation_gaps  
3203:      )  
3204:      simulation_results = self._execute_method(  
3205:          "PDET MunicipalPlanAnalyzer", "_simulate_intervention", context,  
3206:          remediations=remediations  
3207:      )  
3208:  
3209:      # Step 4: Counterfactual analysis for problem dimensioning  
3210:      counterfactual = self._execute_method(  
3211:          "BayesianCounterfactualAuditor", "counterfactual_query", context  
3212:      )  
3213:      effect_stability = self._execute_method(  
3214:          "BayesianCounterfactualAuditor", "test_effect_stability", context,  
3215:          counterfactual=counterfactual  
3216:      )  
3217:  
3218:      # Step 5: Detect numerical inconsistencies  
3219:      numerical_inconsistencies = self._execute_method(  
3220:          "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context  
3221:      )  
3222:      divergence_calc = self._execute_method(  
3223:          "PolicyContradictionDetector", "_calculate_numerical_divergence", context,  
3224:          inconsistencies=numerical_inconsistencies  
3225:      )  
3226:  
3227:      # Step 6: Calculate confidence and analyze performance  
3228:      posterior_confidence = self._execute_method(  
3229:          "BayesianConfidenceCalculator", "calculate_posterior", context,  
3230:          evidence=direct_evidence_audit  
3231:      )  
3232:      performance_analysis = self._execute_method(  
3233:          "PerformanceAnalyzer", "analyze_performance", context  
3234:      )  
3235:  
3236:      raw_evidence = {  
3237:          "magnitude_indicators": {  
3238:              "allocation_gaps": allocation_gaps,  
3239:              "mechanism_gaps": mechanism_gaps,  
3240:              "numerical_inconsistencies": numerical_inconsistencies  
3241:          },  
3242:          "deficit_quantification": divergence_calc,  
3243:          "counterfactual_analysis": counterfactual,  
3244:          "effect_stability": effect_stability,  
3245:          "data_limitations": {
```

```

3246:             "evidence_gaps": direct_evidence_audit.get("gaps", []),
3247:             "systemic_risks": systemic_risk_audit
3248:         },
3249:         "simulation_results": simulation_results,
3250:         "confidence_scores": posterior_confidence,
3251:         "performance_metrics": performance_analysis
3252:     }
3253:
3254:     return {
3255:         "executor_id": self.executor_id,
3256:         "raw_evidence": raw_evidence,
3257:         "metadata": {
3258:             "methods_executed": [log["method"] for log in self.execution_log],
3259:             "gaps_identified": len(allocation_gaps or []) + len(mechanism_gaps or []),
3260:             "inconsistencies_found": len(numerical_inconsistencies or [])
3261:         },
3262:         "execution_metrics": {
3263:             "methods_count": len(self.execution_log),
3264:             "all_succeeded": all(log["success"] for log in self.execution_log)
3265:         }
3266:     }
3267:
3268:
3269: class D1_Q3_BudgetAllocationTracer(BaseExecutor):
3270:     """
3271:     Traces monetary resources assigned to programs in Investment Plan (PPI).
3272:
3273:     Methods (from D1-Q3):
3274:     - FinancialAuditor.trace_financial_allocation
3275:     - FinancialAuditor._process_financial_table
3276:     - FinancialAuditor._match_program_to_node
3277:     - FinancialAuditor._match_goal_to_budget
3278:     - FinancialAuditor._perform_counterfactual_budget_check
3279:     - FinancialAuditor._calculate_sufficiency
3280:     - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility
3281:     - PDET MunicipalPlanAnalyzer._extract_budget_for_pillar
3282:     - PDET MunicipalPlanAnalyzer._identify_funding_source
3283:     - PDET MunicipalPlanAnalyzer._classify_tables
3284:     - PDET MunicipalPlanAnalyzer._analyze_funding_sources
3285:     - PDET MunicipalPlanAnalyzer._score_financial_component
3286:     - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
3287:
3288:
3289:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3290:         raw_evidence = {}
3291:
3292:         # Step 1: Trace complete financial allocation chain
3293:         allocation_trace = self._execute_method(
3294:             "FinancialAuditor", "trace_financial_allocation", context
3295:         )
3296:         processed_tables = self._execute_method(
3297:             "FinancialAuditor", "_process_financial_table", context
3298:         )
3299:
3300:         # Step 2: Match programs to budget nodes
3301:         program_matches = self._execute_method(

```

```
3302:         "FinancialAuditor", "_match_program_to_node", context,
3303:         tables=processed_tables
3304:     )
3305:     goal_budget_matches = self._execute_method(
3306:         "FinancialAuditor", "_match_goal_to_budget", context,
3307:         programs=program_matches
3308:     )
3309:
3310:     # Step 3: Counterfactual checks and sufficiency calculation
3311:     counterfactual_check = self._execute_method(
3312:         "FinancialAuditor", "_perform_counterfactual_budget_check", context,
3313:         matches=goal_budget_matches
3314:     )
3315:     sufficiency_calc = self._execute_method(
3316:         "FinancialAuditor", "_calculate_sufficiency", context,
3317:         allocation=allocation_trace
3318:     )
3319:
3320:     # Step 4: Analyze financial feasibility
3321:     feasibility_analysis = self._execute_method(
3322:         "PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility", context
3323:     )
3324:     pillar_budgets = self._execute_method(
3325:         "PDET MunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
3326:     )
3327:     funding_sources = self._execute_method(
3328:         "PDET MunicipalPlanAnalyzer", "_identify_funding_source", context
3329:     )
3330:
3331:     # Step 5: Classify and analyze tables
3332:     table_classification = self._execute_method(
3333:         "PDET MunicipalPlanAnalyzer", "_classify_tables", context,
3334:         tables=processed_tables
3335:     )
3336:     funding_analysis = self._execute_method(
3337:         "PDET MunicipalPlanAnalyzer", "_analyze_funding_sources", context,
3338:         sources=funding_sources
3339:     )
3340:     financial_score = self._execute_method(
3341:         "PDET MunicipalPlanAnalyzer", "_score_financial_component", context,
3342:         analysis=funding_analysis
3343:     )
3344:
3345:     # Step 6: Aggregate risk and prioritize
3346:     risk_aggregation = self._execute_method(
3347:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
3348:         sufficiency=sufficiency_calc,
3349:         counterfactual=counterfactual_check
3350:     )
3351:
3352:     raw_evidence = {
3353:         "budget_allocations": allocation_trace,
3354:         "program_mappings": program_matches,
3355:         "goal_budget_links": goal_budget_matches,
3356:         "counterfactual_budget_check": counterfactual_check,
3357:         "sufficiency_analysis": sufficiency_calc,
```

```
3358:         "pillar_budgets": pillar_budgets,
3359:         "funding_sources": funding_sources,
3360:         "financial_feasibility": feasibility_analysis,
3361:         "financial_score": financial_score,
3362:         "table_classification": table_classification,
3363:         "funding_analysis": funding_analysis,
3364:         "risk_priorities": risk_aggregation
3365:     }
3366:
3367:     return {
3368:         "executor_id": self.executor_id,
3369:         "raw_evidence": raw_evidence,
3370:         "metadata": {
3371:             "methods_executed": [log["method"] for log in self.execution_log],
3372:             "programs_traced": len(program_matches or []),
3373:             "funding_sources_identified": len(funding_sources or [])
3374:         },
3375:         "execution_metrics": {
3376:             "methods_count": len(self.execution_log),
3377:             "all_succeeded": all(log["success"] for log in self.execution_log)
3378:         }
3379:     }
3380:
3381:
3382: class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
3383:     """
3384:         Identifies installed capacity (entities, staff, equipment) and limitations.
3385:
3386:         Methods (from D1-Q4):
3387:         - PDET Municipal Plan Analyzer .identify_responsible_entities
3388:         - PDET Municipal Plan Analyzer ._extract_entities_ner
3389:         - PDET Municipal Plan Analyzer ._extract_entities_syntax
3390:         - PDET Municipal Plan Analyzer ._classify_entity_type
3391:         - PDET Municipal Plan Analyzer ._score_entity_specificity
3392:         - PDET Municipal Plan Analyzer ._consolidate_entities
3393:         - MechanismPartExtractor.extract_entity_activity
3394:         - MechanismPartExtractor._normalize_entity
3395:         - MechanismPartExtractor._validate_entity_activity
3396:         - MechanismPartExtractor._calculate_ea_confidence
3397:         - OperationalizationAuditor.audit_evidence_traceability
3398:     """
3399:
3400:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3401:         raw_evidence = {}
3402:
3403:         # Step 1: Identify all responsible entities
3404:         entities_identified = self._execute_method(
3405:             "PDET Municipal Plan Analyzer", "identify_responsible_entities", context
3406:         )
3407:
3408:         # Step 2: Extract entities using NER and syntax
3409:         ner_entities = self._execute_method(
3410:             "PDET Municipal Plan Analyzer", "_extract_entities_ner", context
3411:         )
3412:         syntax_entities = self._execute_method(
3413:             "PDET Municipal Plan Analyzer", "_extract_entities_syntax", context
```

```
3414:     )
3415:
3416:     # Step 3: Classify and score entities
3417:     entity_types = self._execute_method(
3418:         "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
3419:         entities=ner_entities + syntax_entities
3420:     )
3421:     specificity_scores = self._execute_method(
3422:         "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
3423:         entities=entity_types
3424:     )
3425:     consolidated = self._execute_method(
3426:         "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
3427:         entities=entity_types
3428:     )
3429:
3430:     # Step 4: Extract entity-activity relationships
3431:     entity_activities = self._execute_method(
3432:         "MechanismPartExtractor", "extract_entity_activity", context,
3433:         entities=consolidated
3434:     )
3435:     normalized = self._execute_method(
3436:         "MechanismPartExtractor", "_normalize_entity", context,
3437:         activities=entity_activities
3438:     )
3439:     validated = self._execute_method(
3440:         "MechanismPartExtractor", "_validate_entity_activity", context,
3441:         normalized=normalized
3442:     )
3443:     ea_confidence = self._execute_method(
3444:         "MechanismPartExtractor", "_calculate_ea_confidence", context,
3445:         validated=validated
3446:     )
3447:
3448:     # Step 5: Audit evidence traceability
3449:     traceability_audit = self._execute_method(
3450:         "OperationalizationAuditor", "audit_evidence_traceability", context,
3451:         entity_activities=validated
3452:     )
3453:
3454:     raw_evidence = {
3455:         "entities_identified": consolidated,
3456:         "entity_types": entity_types,
3457:         "specificity_scores": specificity_scores,
3458:         "entity_activities": validated,
3459:         "activity_confidence": ea_confidence,
3460:         "capacity_indicators": {
3461:             "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
3462:             "equipment_mentions": [e for e in consolidated if e.get("type") == "equipment"],
3463:             "organizational_units": [e for e in consolidated if e.get("type") == "organization"]
3464:         },
3465:         "limitations_identified": (traceability_audit or {}).get("gaps", []),
3466:         "traceability_audit": traceability_audit
3467:     }
3468:
3469:     return {
```

```
3470:         "executor_id": self.executor_id,
3471:         "raw_evidence": raw_evidence,
3472:         "metadata": {
3473:             "methods_executed": [log["method"] for log in self.execution_log],
3474:             "entities_count": len(consolidated or []),
3475:             "activities_extracted": len(validated or [])
3476:         },
3477:         "execution_metrics": {
3478:             "methods_count": len(self.execution_log),
3479:             "all_succeeded": all(log["success"] for log in self.execution_log)
3480:         }
3481:     }
3482:
3483:
3484: class D1_Q5_ScopeJustificationValidator(BaseExecutor):
3485:     """
3486:         Validates scope justification via legal framework and constraint recognition.
3487:
3488:         Methods (from D1-Q5):
3489:         - TemporalLogicVerifier._check_deadline_constraints
3490:         - TemporalLogicVerifier.verify_temporal_consistency
3491:         - CausalInferenceSetup.identify_failure_points
3492:         - CausalExtractor._assess_temporal_coherence
3493:         - TextMiningEngine._analyze_link_text
3494:         - IndustrialPolicyProcessor._analyze_causal_dimensions
3495:         - IndustrialPolicyProcessor._extract_metadata
3496:     """
3497:
3498:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3499:         raw_evidence = {}
3500:
3501:         # Step 1: Verify temporal constraints
3502:         deadline_constraints = self._execute_method(
3503:             "TemporalLogicVerifier", "_check_deadline_constraints", context
3504:         )
3505:         temporal_consistency = self._execute_method(
3506:             "TemporalLogicVerifier", "verify_temporal_consistency", context
3507:         )
3508:
3509:         # Step 2: Identify failure points in scope
3510:         failure_points = self._execute_method(
3511:             "CausalInferenceSetup", "identify_failure_points", context
3512:         )
3513:
3514:         # Step 3: Assess temporal coherence
3515:         temporal_coherence = self._execute_method(
3516:             "CausalExtractor", "_assess_temporal_coherence", context
3517:         )
3518:
3519:         # Step 4: Analyze link text for justifications
3520:         link_analysis = self._execute_method(
3521:             "TextMiningEngine", "_analyze_link_text", context
3522:         )
3523:
3524:         # Step 5: Analyze causal dimensions and extract metadata
3525:         causal_dimensions = self._execute_method()
```

```
3526:         "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
3527:     )
3528:     metadata_extracted = self._execute_method(
3529:         "IndustrialPolicyProcessor", "_extract_metadata", context,
3530:         dimensions=causal_dimensions
3531:     )
3532:
3533:     raw_evidence = {
3534:         "legal_framework_citations": metadata_extracted.get("legal_refs", []),
3535:         "temporal_constraints": {
3536:             "deadline_checks": deadline_constraints,
3537:             "consistency": temporal_consistency,
3538:             "coherence": temporal_coherence
3539:         },
3540:         "budgetary_constraints": metadata_extracted.get("budget_limits", []),
3541:         "competence_constraints": metadata_extracted.get("competence_refs", []),
3542:         "failure_points": failure_points,
3543:         "scope_justifications": (link_analysis or {}).get("justifications", []),
3544:         "causal_dimensions": causal_dimensions
3545:     }
3546:
3547:     return {
3548:         "executor_id": self.executor_id,
3549:         "raw_evidence": raw_evidence,
3550:         "metadata": {
3551:             "methods_executed": [log["method"] for log in self.execution_log],
3552:             "constraints_identified": len(deadline_constraints or []),
3553:             "legal_citations": len(metadata_extracted.get("legal_refs", []))
3554:         },
3555:         "execution_metrics": {
3556:             "methods_count": len(self.execution_log),
3557:             "all_succeeded": all(log["success"] for log in self.execution_log)
3558:         }
3559:     }
3560:
3561:
3562: # =====
3563: # DIMENSION 2: ACTIVITY DESIGN
3564: # =====
3565:
3566: class D2_Q1_StructuredPlanningValidator(BaseExecutor):
3567:     """
3568:         Validates structured format of activities (table/matrix with required columns).
3569:
3570:         Methods (from D2-Q1):
3571:         - PDFProcessor.extract_tables
3572:         - FinancialAuditor._process_financial_table
3573:         - PDET MunicipalPlanAnalyzer._deduplicate_tables
3574:         - PDET MunicipalPlanAnalyzer._classify_tables
3575:         - PDET MunicipalPlanAnalyzer._is_likely_header
3576:         - PDET MunicipalPlanAnalyzer._clean_dataframe
3577:         - ReportingEngine.generate_accountability_matrix
3578:     """
3579:
3580:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3581:         raw_evidence = {}
```

```
3582:  
3583:     # Step 1: Extract all tables  
3584:     extracted_tables = self._execute_method(  
3585:         "PDFProcessor", "extract_tables", context  
3586:     )  
3587:  
3588:     # Step 2: Process financial tables  
3589:     processed_tables = self._execute_method(  
3590:         "FinancialAuditor", "_process_financial_table", context,  
3591:         tables=extracted_tables  
3592:     )  
3593:  
3594:     # Step 3: Deduplicate and classify tables  
3595:     deduplicated = self._execute_method(  
3596:         "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,  
3597:         tables=processed_tables  
3598:     )  
3599:     classified = self._execute_method(  
3600:         "PDETMunicipalPlanAnalyzer", "_classify_tables", context,  
3601:         tables=deduplicated  
3602:     )  
3603:  
3604:     # Step 4: Identify headers and clean dataframes  
3605:     header_checks = self._execute_method(  
3606:         "PDETMunicipalPlanAnalyzer", "is_likely_header", context,  
3607:         tables=classified  
3608:     )  
3609:     cleaned = self._execute_method(  
3610:         "PDETMunicipalPlanAnalyzer", "_clean_dataframe", context,  
3611:         tables=classified  
3612:     )  
3613:  
3614:     # Step 5: Generate accountability matrix  
3615:     accountability_matrix = self._execute_method(  
3616:         "ReportingEngine", "generate_accountability_matrix", context,  
3617:         tables=cleaned  
3618:     )  
3619:  
3620:     raw_evidence = {  
3621:         "tables_extracted": len(extracted_tables),  
3622:         "activity_tables": [t for t in classified if t.get("type") == "activity"],  
3623:         "matrix_structure": accountability_matrix,  
3624:         "required_columns_present": {  
3625:             "responsible_entity": any("responsible" in str(t.get("columns", [])).lower()  
3626:                                     for t in cleaned),  
3627:             "deliverable": any("deliverable" in str(t.get("columns", [])).lower()  
3628:                               for t in cleaned),  
3629:             "timeline": any("timeline" in str(t.get("columns", [])).lower()  
3630:                           for t in cleaned),  
3631:             "cost": any("cost" in str(t.get("columns", [])).lower()  
3632:                           for t in cleaned)  
3633:         },  
3634:         "table_quality": {  
3635:             "clean_tables": len(cleaned),  
3636:             "with_headers": sum(1 for h in header_checks if h  
3637:         }  
3638:
```

```
3638:         }
3639:
3640:     return {
3641:         "executor_id": self.executor_id,
3642:         "raw_evidence": raw_evidence,
3643:         "metadata": {
3644:             "methods_executed": [log["method"] for log in self.execution_log],
3645:             "total_tables": len(extracted_tables),
3646:             "activity_tables": len([t for t in classified if t.get("type") == "activity"])
3647:         },
3648:         "execution_metrics": {
3649:             "methods_count": len(self.execution_log),
3650:             "all_succeeded": all(log["success"] for log in self.execution_log)
3651:         }
3652:     }
3653:
3654:
3655: class D2_Q2_InterventionLogicInferencer(BaseExecutor):
3656:     """
3657:     Infers intervention logic: instrument (how), target (who), causality (why).
3658:
3659:     Methods (from D2-Q2):
3660:     - BayesianMechanismInference.infer_mechanisms
3661:     - BayesianMechanismInference._infer_single_mechanism
3662:     - BayesianMechanismInference._infer_mechanism_type
3663:     - BayesianMechanismInference._test_sufficiency
3664:     - BayesianMechanismInference._test_necessity
3665:     - CausalExtractor.extract_causal_hierarchy
3666:     - TeoriaCambio.construir_grafo_causal
3667:     - TeoriaCambio._esConexion_valida
3668:     - PDET MunicipalPlanAnalyzer.construct_causal_dag
3669:     - BeachEvidentialTest.classify_test
3670:     - IndustrialPolicyProcessor._analyze_causal_dimensions
3671:
3672:
3673:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3674:         raw_evidence = {}
3675:
3676:         # Step 1: Infer mechanisms
3677:         mechanisms = self._execute_method(
3678:             "BayesianMechanismInference", "infer_mechanisms", context
3679:         )
3680:         single_mechanisms = []
3681:         for mech in mechanisms:
3682:             single = self._execute_method(
3683:                 "BayesianMechanismInference", "_infer_single_mechanism", context,
3684:                 mechanism=mech
3685:             )
3686:             single_mechanisms.append(single)
3687:
3688:         mechanism_types = self._execute_method(
3689:             "BayesianMechanismInference", "_infer_mechanism_type", context,
3690:             mechanisms=single_mechanisms
3691:         )
3692:
3693:         # Step 2: Test sufficiency and necessity
```

```
3694:     sufficiency_tests = self._execute_method(
3695:         "BayesianMechanismInference", "_test_sufficiency", context,
3696:         mechanisms=single_mechanisms
3697:     )
3698:     necessity_tests = self._execute_method(
3699:         "BayesianMechanismInference", "_test_necessity", context,
3700:         mechanisms=single_mechanisms
3701:     )
3702:
3703:     # Step 3: Extract causal hierarchy
3704:     causal_hierarchy = self._execute_method(
3705:         "CausalExtractor", "extract_causal_hierarchy", context
3706:     )
3707:
3708:     # Step 4: Build causal graph
3709:     causal_graph = self._execute_method(
3710:         "TeoriaCambio", "construir_grafo_causal", context,
3711:         hierarchy=causal_hierarchy
3712:     )
3713:     connection_validation = self._execute_method(
3714:         "TeoriaCambio", "esConexionValida", context,
3715:         graph=causal_graph
3716:     )
3717:
3718:     # Step 5: Construct DAG
3719:     causal_dag = self._execute_method(
3720:         "PDET Municipal Plan Analyzer", "construct_causal_dag", context,
3721:         graph=causal_graph
3722:     )
3723:
3724:     # Step 6: Classify evidential tests
3725:     evidential_tests = self._execute_method(
3726:         "BeachEvidentialTest", "classify_test", context,
3727:         mechanisms=single_mechanisms
3728:     )
3729:
3730:     # Step 7: Analyze causal dimensions
3731:     causal_dimensions = self._execute_method(
3732:         "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
3733:     )
3734:
3735:     raw_evidence = {
3736:         "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
3737:         "target_populations": [m.get("target") for m in single_mechanisms],
3738:         "causal_logic": {
3739:             "mechanisms": single_mechanisms,
3740:             "mechanism_types": mechanism_types,
3741:             "sufficiency": sufficiency_tests,
3742:             "necessity": necessity_tests
3743:         },
3744:         "causal_hierarchy": causal_hierarchy,
3745:         "causal_graph": causal_graph,
3746:         "causal_dag": causal_dag,
3747:         "evidential_strength": evidential_tests,
3748:         "connection_validation": connection_validation,
3749:         "dimensions": causal_dimensions
```

```
3750:         }
3751:
3752:     return {
3753:         "executor_id": self.executor_id,
3754:         "raw_evidence": raw_evidence,
3755:         "metadata": {
3756:             "methods_executed": [log["method"] for log in self.execution_log],
3757:             "mechanisms_identified": len(single_mechanisms or []),
3758:             "instruments_found": len([m for m in single_mechanisms if m.get("instrument")]),
3759:             "connections_valid": bool(connection_validation)
3760:         },
3761:         "execution_metrics": {
3762:             "methods_count": len(self.execution_log),
3763:             "all_succeeded": all(log["success"] for log in self.execution_log)
3764:         }
3765:     }
3766:
3767:
3768: class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
3769:     """
3770:         Analyzes linkage between activities and root causes/structural determinants.
3771:
3772:     Methods (from D2-Q3):
3773:     - CausalExtractor._extract_causal_links
3774:     - CausalExtractor._calculate_composite_likelihood
3775:     - CausalExtractor._initialize_prior
3776:     - CausalExtractor._calculate_type_transition_prior
3777:     - PDET Municipal Plan Analyzer._identify_causal_edges
3778:     - PDET Municipal Plan Analyzer._refine_edge_probabilities
3779:     - BayesianCounterfactualAuditor.construct_scm
3780:     - BayesianCounterfactualAuditor._create_default_equations
3781:     - SemanticAnalyzer.extract_semantic_cube
3782:     """
3783:
3784:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3785:         raw_evidence = {}
3786:
3787:         # Step 1: Extract causal links
3788:         causal_links = self._execute_method(
3789:             "CausalExtractor", "_extract_causal_links", context
3790:         )
3791:
3792:         # Step 2: Calculate likelihoods
3793:         composite_likelihood = self._execute_method(
3794:             "CausalExtractor", "_calculate_composite_likelihood", context,
3795:             links=causal_links
3796:         )
3797:         prior_init = self._execute_method(
3798:             "CausalExtractor", "_initialize_prior", context
3799:         )
3800:         type_transition_prior = self._execute_method(
3801:             "CausalExtractor", "_calculate_type_transition_prior", context,
3802:             links=causal_links
3803:         )
3804:
3805:         # Step 3: Identify and refine causal edges
```

```
3806:     causal_edges = self._execute_method(
3807:         "PDET MunicipalPlanAnalyzer", "_identify_causal_edges", context,
3808:         links=causal_links
3809:     )
3810:     refined_probabilities = self._execute_method(
3811:         "PDET MunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
3812:         edges=causal_edges
3813:     )
3814:
3815:     # Step 4: Construct structural causal model
3816:     scm = self._execute_method(
3817:         "BayesianCounterfactualAuditor", "construct_scm", context,
3818:         edges=refined_probabilities
3819:     )
3820:     default_equations = self._execute_method(
3821:         "BayesianCounterfactualAuditor", "_create_default_equations", context,
3822:         scm=scm
3823:     )
3824:
3825:     # Step 5: Extract semantic cube
3826:     semantic_cube = self._execute_method(
3827:         "SemanticAnalyzer", "extract_semantic_cube", context
3828:     )
3829:
3830:     raw_evidence = {
3831:         "root_causes_identified": [link.get("root_cause") for link in (causal_links or [])],
3832:         "activity_linkages": causal_links,
3833:         "link_probabilities": refined_probabilities,
3834:         "composite_likelihood": composite_likelihood,
3835:         "prior_INITIALIZATION": prior_init,
3836:         "type_transition_prior": type_transition_prior,
3837:         "structural_model": scm,
3838:         "model_equations": default_equations,
3839:         "semantic_relationships": semantic_cube,
3840:         "determinants_addressed": [link for link in (causal_links or []) if link.get("addresses_determinant")]
3841:     }
3842:
3843:     return {
3844:         "executor_id": self.executor_id,
3845:         "raw_evidence": raw_evidence,
3846:         "metadata": {
3847:             "methods_executed": [log["method"] for log in self.execution_log],
3848:             "causal_links_found": len(causal_links or []),
3849:             "root_causes_count": len(set(link.get("root_cause") for link in (causal_links or [])))
3850:         },
3851:         "execution_metrics": {
3852:             "methods_count": len(self.execution_log),
3853:             "all_succeeded": all(log["success"] for log in self.execution_log)
3854:         }
3855:     }
3856:
3857:
3858: class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
3859:     """
3860:         Identifies implementation risks and mitigation measures.
3861:
```

```
3862:     Methods (from D2-Q4):
3863:     - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
3864:     - PDETMunicipalPlanAnalyzer.sensitivity_analysis
3865:     - PDETMunicipalPlanAnalyzer._interpret_risk
3866:     - PDETMunicipalPlanAnalyzer._compute_robustness_value
3867:     - PDETMunicipalPlanAnalyzer._compute_e_value
3868:     - PDETMunicipalPlanAnalyzer._interpret_sensitivity
3869:     - OperationalizationAuditor._audit_systemic_risk
3870:     - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
3871:     - BayesianCounterfactualAuditor.refutation_and_sanity_checks
3872:     - AdaptivePriorCalculator.sensitivity_analysis
3873: """
3874:
3875: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3876:     raw_evidence = {}
3877:
3878:     # Step 1: Bayesian risk inference
3879:     risk_inference = self._execute_method(
3880:         "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
3881:     )
3882:
3883:     # Step 2: Sensitivity analysis
3884:     sensitivity = self._execute_method(
3885:         "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
3886:         risks=risk_inference
3887:     )
3888:
3889:     # Step 3: Risk interpretation
3890:     risk_interpretation = self._execute_method(
3891:         "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
3892:         inference=risk_inference
3893:     )
3894:
3895:     # Step 4: Compute robustness metrics
3896:     robustness = self._execute_method(
3897:         "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
3898:         sensitivity=sensitivity
3899:     )
3900:     e_value = self._execute_method(
3901:         "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
3902:         robustness=robustness
3903:     )
3904:     sensitivity_interpretation = self._execute_method(
3905:         "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
3906:         sensitivity=sensitivity
3907:     )
3908:
3909:     # Step 5: Audit systemic risks
3910:     systemic_risk_audit = self._execute_method(
3911:         "OperationalizationAuditor", "_audit_systemic_risk", context
3912:     )
3913:
3914:     # Step 6: Aggregate and prioritize risks
3915:     risk_aggregation = self._execute_method(
3916:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
3917:         risks=risk_inference
```

```
3918:         )
3919:
3920:     # Step 7: Refutation and sanity checks
3921:     refutation_checks = self._execute_method(
3922:         "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
3923:         aggregation=risk_aggregation
3924:     )
3925:
3926:     # Step 8: Additional sensitivity analysis
3927:     adaptive_sensitivity = self._execute_method(
3928:         "AdaptivePriorCalculator", "sensitivity_analysis", context,
3929:         risks=risk_inference
3930:     )
3931:
3932:     raw_evidence = {
3933:         "operational_risks": [r for r in risk_inference if r.get("type") == "operational"],
3934:         "social_risks": [r for r in risk_inference if r.get("type") == "social"],
3935:         "security_risks": [r for r in risk_inference if r.get("type") == "security"],
3936:         "mitigation_measures": (risk_interpretation or {}).get("mitigations", []),
3937:         "risk_priorities": risk_aggregation,
3938:         "robustness_metrics": {
3939:             "robustness_value": robustness,
3940:             "e_value": e_value
3941:         },
3942:         "sensitivity_analysis": sensitivity,
3943:         "systemic_risks": systemic_risk_audit,
3944:         "validation_checks": refutation_checks,
3945:         "sensitivity_interpretation": sensitivity_interpretation,
3946:         "adaptive_sensitivity": adaptive_sensitivity,
3947:         "risk_interpretation": risk_interpretation
3948:     }
3949:
3950:     return {
3951:         "executor_id": self.executor_id,
3952:         "raw_evidence": raw_evidence,
3953:         "metadata": {
3954:             "methods_executed": [log["method"] for log in self.execution_log],
3955:             "risks_identified": len(risk_inference or []),
3956:             "mitigations_proposed": len((risk_interpretation or {}).get("mitigations", []))
3957:         },
3958:         "execution_metrics": {
3959:             "methods_count": len(self.execution_log),
3960:             "all_succeeded": all(log["success"] for log in self.execution_log)
3961:         }
3962:     }
3963:
3964:
3965: class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
3966:     """
3967:     Evaluates strategic coherence: complementarity and logical sequence.
3968:
3969:     Methods (from D2-Q5):
3970:     - PolicyContradictionDetector._detect_logical_incompatibilities
3971:     - PolicyContradictionDetector._calculate_coherence_metrics
3972:     - PolicyContradictionDetector._calculate_objective_alignment
3973:     - PolicyContradictionDetector._calculate_graph_fragmentation
```

```
3974:     - OperationalizationAuditor.audit_sequence_logic
3975:     - BayesianMechanismInference._calculate_coherence_factor
3976:     - PDET MunicipalPlanAnalyzer._score_causal_coherence
3977:     - AdaptivePriorCalculator.calculate_likelihood_adaptativo
3978:     """
3979:
3980:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
3981:         raw_evidence = {}
3982:
3983:         # Step 1: Detect logical incompatibilities
3984:         incompatibilities = self._execute_method(
3985:             "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
3986:         )
3987:
3988:         # Step 2: Calculate coherence metrics
3989:         coherence_metrics = self._execute_method(
3990:             "PolicyContradictionDetector", "_calculate_coherence_metrics", context
3991:         )
3992:         objective_alignment = self._execute_method(
3993:             "PolicyContradictionDetector", "_calculate_objective_alignment", context
3994:         )
3995:         graph_fragmentation = self._execute_method(
3996:             "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
3997:         )
3998:
3999:         # Step 3: Audit sequence logic
4000:         sequence_audit = self._execute_method(
4001:             "OperationalizationAuditor", "audit_sequence_logic", context
4002:         )
4003:
4004:         # Step 4: Calculate coherence factors
4005:         coherence_factor = self._execute_method(
4006:             "BayesianMechanismInference", "_calculate_coherence_factor", context,
4007:             metrics=coherence_metrics
4008:         )
4009:         causal_coherence_score = self._execute_method(
4010:             "PDET MunicipalPlanAnalyzer", "_score_causal_coherence", context
4011:         )
4012:
4013:         # Step 5: Adaptive likelihood calculation
4014:         adaptive_likelihood = self._execute_method(
4015:             "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
4016:             coherence=causal_coherence_score
4017:         )
4018:
4019:         raw_evidence = {
4020:             "complementarity_evidence": coherence_metrics.get("complementarity", []),
4021:             "sequential_logic": sequence_audit,
4022:             "logical_incompatibilities": incompatibilities,
4023:             "coherence_scores": {
4024:                 "overall_coherence": coherence_metrics,
4025:                 "objective_alignment": objective_alignment,
4026:                 "causal_coherence": causal_coherence_score,
4027:                 "coherence_factor": coherence_factor
4028:             },
4029:             "graph_metrics": {
```

```
4030:         "fragmentation": graph_fragmentation
4031:     },
4032:     "adaptive_likelihood": adaptive_likelihood
4033:   }
4034:
4035:   return {
4036:     "executor_id": self.executor_id,
4037:     "raw_evidence": raw_evidence,
4038:     "metadata": {
4039:       "methods_executed": [log["method"] for log in self.execution_log],
4040:       "incompatibilities_found": len(incompatibilities),
4041:       "coherence_score": coherence_metrics.get("score", 0)
4042:     },
4043:     "execution_metrics": {
4044:       "methods_count": len(self.execution_log),
4045:       "all_succeeded": all(log["success"] for log in self.execution_log)
4046:     }
4047:   }
4048:
4049:
4050: # =====
4051: # DIMENSION 3: PRODUCTS & OUTPUTS
4052: # =====
4053:
4054: class D3_Q1_IndicatorQualityValidator(BaseExecutor):
4055:   """
4056:     Validates indicator quality: baseline, target, source of verification.
4057:
4058:     Methods (from D3-Q1):
4059:     - PDETMunicipalPlanAnalyzer._score_indicators
4060:     - OperationalizationAuditor.audit_evidence_traceability
4061:     - CausalInferenceSetup.assign_probative_value
4062:     - BeachEvidentialTest.apply_test_logic
4063:     - TextMiningEngine.diagnose_critical_links
4064:     - IndustrialPolicyProcessor._extract_metadata
4065:     - IndustrialPolicyProcessor._calculate_quality_score
4066:     - AdaptivePriorCalculator.generate_traceability_record
4067:   """
4068:
4069:   def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
4070:     raw_evidence = {}
4071:
4072:     # Step 1: Score indicators
4073:     indicator_scores = self._execute_method(
4074:       "PDETMunicipalPlanAnalyzer", "_score_indicators", context
4075:     )
4076:
4077:     # Step 2: Audit evidence traceability
4078:     traceability_audit = self._execute_method(
4079:       "OperationalizationAuditor", "audit_evidence_traceability", context,
4080:       indicators=indicator_scores
4081:     )
4082:
4083:     # Step 3: Assign probative value
4084:     probative_values = self._execute_method(
4085:       "CausalInferenceSetup", "assign_probative_value", context,
```

```
4086:         indicators=indicator_scores
4087:     )
4088:
4089:     # Step 4: Apply evidential tests
4090:     evidential_tests = self._execute_method(
4091:         "BeachEvidentialTest", "apply_test_logic", context,
4092:         indicators=indicator_scores
4093:     )
4094:
4095:     # Step 5: Diagnose critical links
4096:     critical_links = self._execute_method(
4097:         "TextMiningEngine", "diagnose_critical_links", context
4098:     )
4099:
4100:    # Step 6: Extract and score metadata
4101:    metadata = self._execute_method(
4102:        "IndustrialPolicyProcessor", "_extract_metadata", context
4103:    )
4104:    quality_score = self._execute_method(
4105:        "IndustrialPolicyProcessor", "_calculate_quality_score", context,
4106:        metadata=metadata
4107:    )
4108:
4109:    # Step 7: Generate traceability record
4110:    traceability_record = self._execute_method(
4111:        "AdaptivePriorCalculator", "generate_traceability_record", context,
4112:        indicators=indicator_scores
4113:    )
4114:
4115:    raw_evidence = {
4116:        "indicators_with_baseline": [i for i in indicator_scores if i.get("has_baseline")],
4117:        "indicators_with_target": [i for i in indicator_scores if i.get("has_target")],
4118:        "indicators_with_source": [i for i in indicator_scores if i.get("has_source")],
4119:        "indicator_quality_scores": indicator_scores,
4120:        "traceability": traceability_audit,
4121:        "probative_values": probative_values,
4122:        "evidential_strength": evidential_tests,
4123:        "critical_links": critical_links,
4124:        "overall_quality_score": quality_score,
4125:        "traceability_record": traceability_record
4126:    }
4127:
4128:    return {
4129:        "executor_id": self.executor_id,
4130:        "raw_evidence": raw_evidence,
4131:        "metadata": {
4132:            "methods_executed": [log["method"] for log in self.execution_log],
4133:            "total_indicators": len(indicator_scores or []),
4134:            "complete_indicators": len([i for i in indicator_scores
4135:                if i.get("has_baseline") and i.get("has_target") and i.get("has_source")]),
4136:            "critical_links_assessed": len(critical_links or [])
4137:        },
4138:        "execution_metrics": {
4139:            "methods_count": len(self.execution_log),
4140:            "all_succeeded": all(log["success"] for log in self.execution_log)
4141:        }
4142:    }
```

```
4142:     }
4143:
4144:
4145: class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
4146:     """
4147:         DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY \200\224 Analyzes proportionality of targets to the diagnosed universe using canonical D3 notation.
4148:         Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes tests, and semantic indicator quality.
4149:
4150:     Methods (from D3-Q2):
4151:         - AdvancedDAGValidator._calculate_bayesian_posterior
4152:         - AdvancedDAGValidator._calculate_confidence_interval
4153:         - AdaptivePriorCalculator._adjust_domain_weights
4154:         - PDET MunicipalPlanAnalyzer._get_spanish_stopwords
4155:         - BayesianMechanismInference._log_refactored_components
4156:         - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility
4157:         - PDET MunicipalPlanAnalyzer._score_indicators
4158:         - PDET MunicipalPlanAnalyzer._interpret_risk
4159:         - FinancialAuditor._calculate_sufficiency
4160:         - BayesianMechanismInference._test_sufficiency
4161:         - BayesianMechanismInference._test_necessity
4162:         - PDET MunicipalPlanAnalyzer._assess_financial_sustainability
4163:         - AdaptivePriorCalculator.calculate_likelihood_adaptativo
4164:         - IndustrialPolicyProcessor._calculate_quality_score
4165:         - TeoriaCambio._generar_sugerencias_internas
4166:         - PDET MunicipalPlanAnalyzer._deduplicate_tables
4167:         - PDET MunicipalPlanAnalyzer._indicator_to_dict
4168:         - PDET MunicipalPlanAnalyzer._generate_recommendations
4169:         - IndustrialPolicyProcessor._compile_pattern_registry
4170:         - IndustrialPolicyProcessor._build_point_patterns
4171:         - IndustrialPolicyProcessor._empty_result
4172:     """
4173:
4174:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
4175:         raw_evidence = {}
4176:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
4177:
4178:         # Step 0: Financial feasibility snapshot and indicator quality
4179:         financial_feasibility = self._execute_method(
4180:             "PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility", context
4181:         )
4182:         indicator_quality = self._execute_method(
4183:             "PDET MunicipalPlanAnalyzer", "_score_indicators", context
4184:         )
4185:         spanish_stopwords = self._execute_method(
4186:             "PDET MunicipalPlanAnalyzer", "_get_spanish_stopwords", context
4187:         )
4188:         funding_sources = self._execute_method(
4189:             "PDET MunicipalPlanAnalyzer", "_analyze_funding_sources", context,
4190:             financial_indicators=financial_feasibility.get("financial_indicators", []),
4191:             tables=context.get("tables", [])
4192:         )
4193:         financial_component = self._execute_method(
4194:             "PDET MunicipalPlanAnalyzer", "_score_financial_component", context,
4195:             financial_analysis=financial_feasibility
4196:         )
4197:         pattern_registry = self._execute_method(
```

```
4198:         "IndustrialPolicyProcessor", "_compile_pattern_registry", context
4199:     )
4200:     point_patterns = self._execute_method(
4201:         "IndustrialPolicyProcessor", "_build_point_patterns", context
4202:     )
4203:     empty_policy_result = self._execute_method(
4204:         "IndustrialPolicyProcessor", "_empty_result", context
4205:     )
4206:     dedup_tables = self._execute_method(
4207:         "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
4208:         tables=context.get("tables", [])
4209:     )
4210:     # Type-safe indicator extraction: explicit None, not wrong-typed {}
4211:     first_indicator = None
4212:     if isinstance(financial_feasibility.get("financial_indicators", []), list):
4213:         inds = financial_feasibility.get("financial_indicators", [])
4214:         if inds and isinstance(inds[0], dict):
4215:             first_indicator = inds[0]
4216:
4217:     # Pass None explicitly when no indicator exists, maintaining type contract
4218:     indicator_dict = None
4219:     if first_indicator is not None:
4220:         indicator_dict = self._execute_method(
4221:             "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
4222:             ind=first_indicator
4223:         )
4224:     proportionality_recommendations = self._execute_method(
4225:         "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
4226:         analysis_results={
4227:             "financial_analysis": financial_feasibility,
4228:             "quality_score": quality_score
4229:         }
4230:     )
4231:
4232:     # Step 1: Calculate sufficiency
4233:     sufficiency_calc = self._execute_method(
4234:         "FinancialAuditor", "_calculate_sufficiency", context
4235:     )
4236:
4237:     # Step 2: Test sufficiency and necessity of targets
4238:     sufficiency_test = self._execute_method(
4239:         "BayesianMechanismInference", "_test_sufficiency", context
4240:     )
4241:     necessity_test = self._execute_method(
4242:         "BayesianMechanismInference", "_test_necessity", context
4243:     )
4244:
4245:     # Step 3: Assess financial sustainability
4246:     sustainability_assessment = self._execute_method(
4247:         "PDETMunicipalPlanAnalyzer", "_assess_financial_sustainability", context
4248:     )
4249:     risk_interpretation = self._execute_method(
4250:         "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
4251:         risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)
4252:     )
4253:
```

```
4254:     # Step 4: Calculate adaptive likelihood
4255:     adaptive_likelihood = self._execute_method(
4256:         "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context
4257:     )
4258:     domain_scores = {
4259:         "structural": sufficiency_calc.get("coverage_ratio", 0.0),
4260:         "financial": financial_feasibility.get("sustainability_score", 0.0),
4261:         "semantic": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0
4262:     }
4263:     adjusted_weights = self._execute_method(
4264:         "AdaptivePriorCalculator", "_adjust_domain_weights", context,
4265:         domain_scores=domain_scores
4266:     )
4267:     avg_confidence = self._execute_method(
4268:         "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
4269:         dimension_analysis={"D3": {"dimension_confidence": domain_scores.get("structural", 0.0)}}
4270:     )
4271:
4272:     # Step 6: Generate internal suggestions
4273:     internal_suggestions = self._execute_method(
4274:         "TeoriaCambio", "_generar_sugerencias_internas", context
4275:     )
4276:     # Bayesian posterior diagnostics for proportionality evidence
4277:     posterior_probability = self._execute_method(
4278:         "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,
4279:         likelihood=sufficiency_calc.get("coverage_ratio", 0.5),
4280:         prior=0.5
4281:     )
4282:     confidence_interval = self._execute_method(
4283:         "AdvancedDAGValidator", "_calculate_confidence_interval", context,
4284:         s=int(sufficiency_calc.get("covered_targets", 0)),
4285:         n=max(1, int(sufficiency_calc.get("targets_total", len(context.get("product_targets", []))))),
4286:         conf=0.95
4287:     )
4288:     self._execute_method(
4289:         "BayesianMechanismInference", "_log_refactored_components", context
4290:     )
4291:
4292:     raw_evidence = {
4293:         "target_population_size": context.get("diagnosed_universe", 0),
4294:         "product_targets": context.get("product_targets", []),
4295:         "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
4296:         "dosage_analysis": sufficiency_calc.get("dosage", {}),
4297:         "sufficiency_test": sufficiency_test,
4298:         "necessity_test": necessity_test,
4299:         "sustainability": sustainability_assessment,
4300:         "financial_feasibility": financial_feasibility,
4301:         "indicator_quality": indicator_quality,
4302:         "risk_interpretation": risk_interpretation,
4303:         "proportionality_score": quality_score,
4304:         "recommendations": internal_suggestions,
4305:         "stopwords_spanish": spanish_stopwords,
4306:         "funding_sources_analysis": funding_sources,
4307:         "financial_component_score": financial_component,
4308:         "pattern_registry": pattern_registry,
4309:         "point_patterns": point_patterns,
```

```

4310:         "empty_policy_result": empty_policy_result,
4311:         "avg_confidence": avg_confidence,
4312:         "deduplicated_tables": dedup_tables,
4313:         "indicator_sample": indicator_dict,
4314:         "proportionality_recommendations": proportionality_recommendations,
4315:         "adjusted_domain_weights": adjusted_weights,
4316:         "posterior_proportionality": posterior_probability,
4317:         "coverage_interval": confidence_interval
4318:     }
4319:
4320:     return {
4321:         "executor_id": self.executor_id,
4322:         "raw_evidence": raw_evidence,
4323:         "metadata": {
4324:             "methods_executed": [log["method"] for log in self.execution_log],
4325:             "targets_analyzed": len(context.get("product_targets", [])),
4326:             "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
4327:             "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
4328:             "dimension_code": dim_info.code,
4329:             "dimension_label": dim_info.label
4330:         },
4331:         "execution_metrics": {
4332:             "methods_count": len(self.execution_log),
4333:             "all_succeeded": all(log["success"] for log in self.execution_log)
4334:         }
4335:     }
4336:
4337:
4338: class D3_Q3_TraceabilityValidator(BaseExecutor):
4339:     """
4340:     DIM03_Q03_TRACEABILITY_BUDGET_ORG \200\224 Validates budgetary and organizational traceability of products under canonical D3 notation.
4341:     Epistemic mix: structural budget tracing, organizational semantics, and accountability synthesis.
4342:
4343:     Methods executed (in order):
4344:     Step 1: Budget matching - FinancialAuditor._match_program_to_node
4345:     Step 2: Goal-budget matching - FinancialAuditor._match_goal_to_budget
4346:     Step 3: Responsibility extraction - PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
4347:     Step 4: Entity consolidation - PDETMunicipalPlanAnalyzer._consolidate_entities
4348:     Step 5: Entity identification - PDETMunicipalPlanAnalyzer.identify_responsible_entities
4349:     Step 6: Clarity scoring - PDETMunicipalPlanAnalyzer._score_responsibility_clarity
4350:     Step 7: Document processing - PolicyAnalysisEmbedder.process_document
4351:     Step 8: Query generation - PolicyAnalysisEmbedder._generate_query_from_pdq
4352:     Step 9: Semantic search - PolicyAnalysisEmbedder.semantic_search
4353:     Step 10: MMR diversification - PolicyAnalysisEmbedder._apply_mmr
4354:     Step 11: Semantic cube baseline - SemanticAnalyzer._empty_semantic_cube
4355:     Step 12: Policy domain classification - SemanticAnalyzer._classify_policy_domain
4356:     Step 13: Cross-cutting themes - SemanticAnalyzer._classify_cross_cutting_themes
4357:     Step 14: Value chain classification - SemanticAnalyzer._classify_value_chain_link
4358:     Step 15: Segment vectorization - SemanticAnalyzer._vectorize_segments
4359:     Step 16: Segment processing - SemanticAnalyzer._process_segment
4360:     Step 17: Semantic complexity - SemanticAnalyzer._calculate_semantic_complexity
4361:     Step 18: Evidence confidence - IndustrialPolicyProcessor._compute_evidence_confidence
4362:     Step 19: Entity serialization - PDETMunicipalPlanAnalyzer._entity_to_dict (loop)
4363:     Step 20: Traceability record - AdaptivePriorCalculator.generate_traceability_record
4364:     Step 21: PDQ report - PolicyAnalysisEmbedder.generate_pdq_report
4365:     Step 22: Accountability matrix - ReportingEngine.generate_accountability_matrix

```

```
4366: """
4367:
4368:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
4369:         raw_evidence = {}
4370:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
4371:         document_text = context.get("document_text", "")
4372:         document_metadata = context.get("metadata", {})
4373:
4374:         # Step 1: Match programs to budget nodes
4375:         program_matches = self._execute_method(
4376:             "FinancialAuditor", "_match_program_to_node", context
4377:         )
4378:         goal_budget_matches = self._execute_method(
4379:             "FinancialAuditor", "_match_goal_to_budget", context,
4380:             programs=program_matches
4381:         )
4382:
4383:         # Step 2: Extract responsibility assignments
4384:         responsibility_data = self._execute_method(
4385:             "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
4386:         )
4387:         consolidated_entities = self._execute_method(
4388:             "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
4389:             entities=responsibility_data
4390:         )
4391:         responsible_entities = self._execute_method(
4392:             "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
4393:         )
4394:         responsibility_clarity = self._execute_method(
4395:             "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
4396:             entities=consolidated_entities
4397:         )
4398:         # Semantic traceability via embeddings
4399:         semantic_chunks = self._execute_method(
4400:             "PolicyAnalysisEmbedder", "process_document", context,
4401:             document_text=document_text,
4402:             document_metadata=document_metadata
4403:         )
4404:         pdq_query = self._execute_method(
4405:             "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
4406:             pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
4407:         )
4408:         semantic_hits = self._execute_method(
4409:             "PolicyAnalysisEmbedder", "semantic_search", context,
4410:             query=pdq_query,
4411:             document_chunks=semantic_chunks or []
4412:         )
4413:         diversified_hits = self._execute_method(
4414:             "PolicyAnalysisEmbedder", "_apply_mmr", context,
4415:             ranked_results=semantic_hits or []
4416:         )
4417:         semantic_cube_stub = self._execute_method(
4418:             "SemanticAnalyzer", "_empty_semantic_cube", context
4419:         )
4420:         domain_scores = self._execute_method(
4421:             "SemanticAnalyzer", "_classify_policy_domain", context,
```

```
4422:         segment=document_text
4423:     )
4424:     cross_cutting = self._execute_method(
4425:         "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
4426:         segment=document_text
4427:     )
4428:     value_chain = self._execute_method(
4429:         "SemanticAnalyzer", "_classify_value_chain_link", context,
4430:         segment=document_text
4431:     )
4432:     semantic_vectors = self._execute_method(
4433:         "SemanticAnalyzer", "_vectorize_segments", context,
4434:         segments=[document_text]
4435:     )
4436:     processed_segment = self._execute_method(
4437:         "SemanticAnalyzer", "_process_segment", context,
4438:         segment=document_text,
4439:         idx=0,
4440:         vector=semantic_vectors[0] if semantic_vectors else None
4441:     )
4442:     semantic_complexity = self._execute_method(
4443:         "SemanticAnalyzer", "_calculate_semantic_complexity", context,
4444:         semantic_cube=semantic_cube_stub
4445:     )
4446:     evidence_confidence = self._execute_method(
4447:         "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
4448:         matches=[m.get("bpin", "") for m in program_matches if isinstance(m, dict)],
4449:         text_length=len(document_text),
4450:         pattern_specificity=0.5
4451:     )
4452:
4453:     consolidated_entities = self._safe_process_list(
4454:         consolidated_entities, label="consolidated_entities"
4455:     )
4456:
4457:     entity_dicts = []
4458:     for e in consolidated_entities[:5]:
4459:         if not (isinstance(e, dict) or hasattr(e, "__dict__")):
4460:             continue
4461:
4462:             try:
4463:                 entity_dict = self._execute_method(
4464:                     "PDET MunicipalPlanAnalyzer", "_entity_to_dict", context, entity=e
4465:                 )
4466:                 entity_dict = self._safe_process_dict(entity_dict, label=f"entity_dict_{len(entity_dicts)}")
4467:                 entity_dicts.append(entity_dict)
4468:             except MemoryError:
4469:                 logger.error("Memory exhausted during entity conversion, stopping")
4470:                 break
4471:             except ExecutorFailure as e:
4472:                 logger.warning(f"Entity conversion failed: {e}")
4473:                 continue
4474:
4475: # Step 3: Generate traceability records
4476: traceability_record = self._execute_method(
4477:     "AdaptivePriorCalculator", "generate_traceability_record", context,
```

```
4478:         matches=program_matches
4479:     )
4480:
4481:     # Step 4: Generate PDQ report
4482:     pdq_report = self._execute_method(
4483:         "PolicyAnalysisEmbedder", "generate_pdq_report", context,
4484:         traceability=traceability_record
4485:     )
4486:
4487:     # Step 5: Generate accountability matrix
4488:     accountability_matrix = self._execute_method(
4489:         "ReportingEngine", "generate_accountability_matrix", context,
4490:         entities=consolidated_entities
4491:     )
4492:
4493:     raw_evidence = {
4494:         "budgetary_traceability": {
4495:             "bpin_codes": [m.get("bpin") for m in (program_matches or []) if m.get("bpin")],
4496:             "project_codes": [m.get("project_code") for m in (program_matches or []) if m.get("project_code")],
4497:             "budget_matches": goal_budget_matches
4498:         },
4499:         "organizational_traceability": {
4500:             "responsible_entities": consolidated_entities,
4501:             "office_assignments": [e for e in (consolidated_entities or []) if e.get("office")],
4502:             "secretariat_assignments": [e for e in (consolidated_entities or []) if e.get("secretariat")]
4503:         },
4504:         "traceability_record": traceability_record,
4505:         "pdq_report": pdq_report,
4506:         "accountability_matrix": accountability_matrix,
4507:         "responsible_entities": responsible_entities,
4508:         "responsibility_clarity_score": responsibility_clarity,
4509:         "semantic_traceability": {
4510:             "query": pdq_query,
4511:             "semantic_hits": semantic_hits,
4512:             "diversified_hits": diversified_hits
4513:         },
4514:         "semantic_cube_baseline": semantic_cube_stub,
4515:         "policy_domain_scores": domain_scores,
4516:         "responsibility_entities_dict": entity_dicts,
4517:         "cross_cutting_themes": cross_cutting,
4518:         "value_chain_links": value_chain,
4519:         "semantic_vectors": semantic_vectors,
4520:         "semantic_complexity": semantic_complexity,
4521:         "evidence_confidence": evidence_confidence,
4522:         "processed_segment": processed_segment
4523:     }
4524:
4525:     return {
4526:         "executor_id": self.executor_id,
4527:         "raw_evidence": raw_evidence,
4528:         "metadata": {
4529:             "methods_executed": [log["method"] for log in self.execution_log],
4530:             "products_with_bpin": len([m for m in program_matches if isinstance(m, dict) and m.get("bpin")]),
4531:             "products_with_responsible": len(consolidated_entities) if consolidated_entities else 0,
4532:             "total_semantic_hits": len(semantic_hits) if semantic_hits else 0,
4533:             "has_semantic_hits": bool(semantic_hits),
```

```

4534:         "total_responsible_entities": len(responsible_entities) if responsible_entities else 0,
4535:         "has_responsible_entities": bool(responsible_entities),
4536:         "total_diversified_hits": len(diversified_hits) if diversified_hits else 0,
4537:         "total_entity_dicts": len(entity_dicts) if entity_dicts else 0,
4538:         "has_semantic_vectors": bool(semantic_vectors),
4539:         "total_semantic_vectors": len(semantic_vectors) if semantic_vectors else 0,
4540:         "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
4541:         "dimension_code": dim_info.code,
4542:         "dimension_label": dim_info.label
4543:     },
4544:     "execution_metrics": {
4545:         "methods_count": len(self.execution_log),
4546:         "all_succeeded": all(log["success"] for log in self.execution_log)
4547:     }
4548: }
4549:
4550:
4551: class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
4552: """
4553: DIM03_Q04_TECHNICAL_FEASIBILITY \200\224 Evaluates activity-product feasibility vs resources/deadlines (canonical D3).
4554: Epistemic mix: structural DAG validity, causal necessity, performance/implementation readiness, and statistical robustness.
4555:
4556: Methods executed (in order):
4557: Step 1: Acyclicity p-value - AdvancedDAGValidator.calculate_acyclicity_pvalue
4558: Step 2: Acyclicity check - AdvancedDAGValidator._is_acyclic
4559: Step 3: Graph statistics - AdvancedDAGValidator.get_graph_stats
4560: Step 4: Node importance - AdvancedDAGValidator._calculate_node_importance
4561: Step 5: Subgraph generation - AdvancedDAGValidator._generate_subgraph
4562: Step 6: Node addition - AdvancedDAGValidator.add_node
4563: Step 7: Edge addition - AdvancedDAGValidator.add_edge
4564: Step 8: Node export - AdvancedDAGValidator.export_nodes
4565: Step 9: RNG initialization - AdvancedDAGValidator._initialize_rng
4566: Step 10: Statistical power - AdvancedDAGValidator._calculate_statistical_power
4567: Step 11: Node validator - AdvancedDAGValidator._get_node_validator
4568: Step 12: Empty result creation - AdvancedDAGValidator._create_empty_result
4569: Step 13: Necessity test - BayesianMechanismInference._test_necessity
4570: Step 14: Validation suite - IndustrialGradeValidator.execute_suite
4571: Step 15: Connection matrix - IndustrialGradeValidator.validate_connection_matrix
4572: Step 16: Performance benchmarks - IndustrialGradeValidator.run_performance_benchmarks
4573: Step 17: Benchmark operation - IndustrialGradeValidator._benchmark_operation
4574: Step 18: Metric logging - IndustrialGradeValidator._log_metric
4575: Step 19: Engine readiness - IndustrialGradeValidator.validate_engine_readiness
4576: Step 20: Performance analysis - PerformanceAnalyzer.analyze_performance
4577: Step 21: Loss functions - PerformanceAnalyzer._calculate_loss_functions
4578: Step 22: Resource likelihood - HierarchicalGenerativeModel._calculate_likelihood
4579: Step 23: ESS calculation - HierarchicalGenerativeModel._calculate_ess
4580: Step 24: R-hat calculation - HierarchicalGenerativeModel._calculate_r_hat
4581: Step 25: Causal categories validation - IndustrialGradeValidator.validate_causal_categories
4582: Step 26: Category extraction - TeoriaCambio._extraer_categorias
4583: """
4584:
4585: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
4586:     raw_evidence = {}
4587:     dim_info = get_dimension_info(CanonicalDimension.D3.value)
4588:     plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")
4589:

```

```
4590:     # Step 1: Validate DAG structure
4591:     acyclicity_pvalue = self._execute_method(
4592:         "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
4593:     )
4594:     is_acyclic = self._execute_method(
4595:         "AdvancedDAGValidator", "_is_acyclic", context
4596:     )
4597:     graph_stats = self._execute_method(
4598:         "AdvancedDAGValidator", "get_graph_stats", context
4599:     )
4600:     node_importance = self._execute_method(
4601:         "AdvancedDAGValidator", "_calculate_node_importance", context
4602:     )
4603:     subgraph = self._execute_method(
4604:         "AdvancedDAGValidator", "_generate_subgraph", context
4605:     )
4606:     added_node = self._execute_method(
4607:         "AdvancedDAGValidator", "add_node", context,
4608:         node_name="temp_node"
4609:     )
4610:     added_edge = self._execute_method(
4611:         "AdvancedDAGValidator", "add_edge", context,
4612:         source="temp_node",
4613:         target="temp_target",
4614:         weight=1.0
4615:     )
4616:     node_export = self._execute_method(
4617:         "AdvancedDAGValidator", "export_nodes", context
4618:     )
4619:     rng_seed = self._execute_method(
4620:         "AdvancedDAGValidator", "_initialize_rng", context,
4621:         plan_name=plan_name,
4622:         salt=dim_info.code
4623:     )
4624:     stat_power = self._execute_method(
4625:         "AdvancedDAGValidator", "calculate_statistical_power", context,
4626:         s=int(graph_stats.get("edges", 0)),
4627:         n=max(1, int(graph_stats.get("nodes", 1)))
4628:     )
4629:     node_validator = self._execute_method(
4630:         "AdvancedDAGValidator", "_get_node_validator", context,
4631:         node_type="producto"
4632:     )
4633:     empty_result = self._execute_method(
4634:         "AdvancedDAGValidator", "_create_empty_result", context,
4635:         plan_name=plan_name,
4636:         seed=rng_seed,
4637:         timestamp=context.get("metadata", {}).get("timestamp", "")
4638:     )
4639:
4640:     # Step 2: Test necessity of activities for products
4641:     necessity_test = self._execute_method(
4642:         "BayesianMechanismInference", "_test_necessity", context
4643:     )
4644:
4645:     # Step 3: Execute industrial-grade validation
```

```
4646:         validation_suite = self._execute_method(
4647:             "IndustrialGradeValidator", "execute_suite", context
4648:         )
4649:         connection_validation = self._execute_method(
4650:             "IndustrialGradeValidator", "validate_connection_matrix", context
4651:         )
4652:         performance_benchmarks = self._execute_method(
4653:             "IndustrialGradeValidator", "run_performance_benchmarks", context
4654:         )
4655:         benchmark_ops = self._execute_method(
4656:             "IndustrialGradeValidator", "_benchmark_operation", context
4657:         )
4658:         metric_log = self._execute_method(
4659:             "IndustrialGradeValidator", "_log_metric", context,
4660:             name="custom_latency",
4661:             value=graph_stats.get("edges", 0),
4662:             unit="edges",
4663:             threshold=10.0
4664:         )
4665:         engine_readiness = self._execute_method(
4666:             "IndustrialGradeValidator", "validate_engine_readiness", context
4667:         )
4668:
4669:     # Step 4: Analyze performance
4670:     performance_analysis = self._execute_method(
4671:         "PerformanceAnalyzer", "analyze_performance", context
4672:     )
4673:     loss_functions = self._execute_method(
4674:         "PerformanceAnalyzer", "_calculate_loss_functions", context
4675:     )
4676:     # Likelihood estimation for resource adequacy
4677:     resource_likelihood = self._execute_method(
4678:         "HierarchicalGenerativeModel", "calculate_likelihood", context,
4679:         mechanism_type="tecnico",
4680:         observations={"coherence": (performance_analysis or {}).get("resource_fit", {}).get("score", 0.0)}
4681:     )
4682:
4683:     # Step 5: Calculate effective sample size
4684:     ess = self._execute_method(
4685:         "HierarchicalGenerativeModel", "_calculate_ess", context
4686:     )
4687:     r_hat = self._execute_method(
4688:         "HierarchicalGenerativeModel", "_calculate_r_hat", context,
4689:         chains=[]
4690:     )
4691:     causal_categories_valid = self._execute_method(
4692:         "IndustrialGradeValidator", "validate_causal_categories", context
4693:     )
4694:     extracted_categories = self._execute_method(
4695:         "TeoriaCambio", "_extraer_categorias", context,
4696:         text=context.get("document_text", "")
4697:     )
4698:
4699:     raw_evidence = {
4700:         "activity_product_mapping": connection_validation,
4701:         "resource_adequacy": (performance_analysis or {}).get("resource_fit", {}),
```

```
4702:         "timeline_feasibility": (performance_analysis or {}).get("timeline_feasibility", {}),
4703:         "technical_validation": {
4704:             "dag_valid": is_acyclic,
4705:             "acyclicity_p": acyclicity_pvalue,
4706:             "necessity_score": necessity_test,
4707:             "graph_stats": graph_stats,
4708:             "node_importance": node_importance,
4709:             "subgraph_sample": subgraph,
4710:             "added_node": added_node,
4711:             "added_edge": added_edge,
4712:             "node_validator": node_validator,
4713:             "empty_result": empty_result,
4714:             "node_export": node_export,
4715:             "rng_seed": rng_seed,
4716:             "statistical_power": stat_power
4717:         },
4718:         "performance_metrics": {
4719:             "benchmarks": performance_benchmarks,
4720:             "loss_functions": loss_functions,
4721:             "ess": ess,
4722:             "r_hat": r_hat,
4723:             "resource_likelihood": resource_likelihood
4724:         },
4725:         "engine_readiness": engine_readiness,
4726:         "feasibility_score": validation_suite.get("overall_score", 0),
4727:         "causal_categories_valid": causal_categories_valid,
4728:         "extracted_categories": extracted_categories,
4729:         "metric_log": metric_log
4730:     }
4731:
4732:     return {
4733:         "executor_id": self.executor_id,
4734:         "raw_evidence": raw_evidence,
4735:         "metadata": {
4736:             "methods_executed": [log["method"] for log in self.execution_log],
4737:             "dag_is_valid": is_acyclic,
4738:             "feasibility_score": (validation_suite or {}).get("overall_score", 0) if validation_suite else 0,
4739:             "total_graph_nodes": (graph_stats or {}).get("nodes", 0) if graph_stats else 0,
4740:             "total_graph_edges": (graph_stats or {}).get("edges", 0) if graph_stats else 0,
4741:             "has_node_export": bool(node_export),
4742:             "total_exported_nodes": len(node_export) if node_export else 0,
4743:             "has_subgraph": bool(subgraph),
4744:             "total_extracted_categories": len(extracted_categories) if extracted_categories else 0,
4745:             "has_extracted_categories": bool(extracted_categories),
4746:             "statistical_power": stat_power if isinstance(stat_power, (int, float)) else 0.0,
4747:             "has_engine_readiness": bool(engine_readiness),
4748:             "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
4749:             "dimension_code": dim_info.code,
4750:             "dimension_label": dim_info.label
4751:         },
4752:         "execution_metrics": {
4753:             "methods_count": len(self.execution_log),
4754:             "all_succeeded": all(log["success"] for log in self.execution_log)
4755:         }
4756:     }
4757:
```

```
4758:  
4759: class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):  
4760:     """  
4761:         DIM03_Q05_OUTPUT_OUTCOME_LINKAGE \200\224 Analyzes mechanisms linking outputs to outcomes with canonical D3 labeling.  
4762:         Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect estimation, and Bayesian mechanism inference.  
4763:  
4764:     Methods (from D3-Q5):  
4765:         - PDET MunicipalPlanAnalyzer._identify_confounders  
4766:         - PDET MunicipalPlanAnalyzer._effect_to_dict  
4767:         - PDET MunicipalPlanAnalyzer._scenario_to_dict  
4768:         - PDET MunicipalPlanAnalyzer._simulate_intervention  
4769:         - PDET MunicipalPlanAnalyzer._generate_recommendations  
4770:         - PDET MunicipalPlanAnalyzer._identify_causal_nodes  
4771:         - BayesianCounterfactualAuditor._evaluate_factual  
4772:         - BayesianCounterfactualAuditor._evaluate_counterfactual  
4773:         - CausalExtractor._assess_financial_consistency  
4774:         - BayesianMechanismInference._infer_activity_sequence  
4775:         - BayesianMechanismInference._generate_necessity_remediation  
4776:         - BayesianCounterfactualAuditor.refutation_and_sanity_checks  
4777:         - IndustrialPolicyProcessor._load_questionnaire  
4778:         - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility  
4779:         - PDET MunicipalPlanAnalyzer.construct_causal_dag  
4780:         - PDET MunicipalPlanAnalyzer.estimate_causal_effects  
4781:         - PDET MunicipalPlanAnalyzer.generate_counterfactuals  
4782:         - CausalExtractor._build_type_hierarchy  
4783:         - CausalExtractor._check_structuralViolation  
4784:         - CausalExtractor._calculate_type_transition_prior  
4785:         - CausalExtractor._calculate_textual_proximity  
4786:         - TeoriaCambio._validar orden_causal  
4787:         - PDET MunicipalPlanAnalyzer._refine_edge_probabilities  
4788:         - PolicyAnalysisEmbedder.compare_policy_interventions  
4789:         - BayesianMechanismInference.infer_mechanisms  
4790:     """  
4791:  
4792:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:  
4793:         raw_evidence = {}  
4794:         dim_info = get_dimension_info(CanonicalDimension.D3.value)  
4795:  
4796:             # Step 0: Build causal backbone and effects  
4797:             financial_analysis = self._execute_method(  
4798:                 "PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility", context  
4799:             )  
4800:             causal_dag = self._execute_method(  
4801:                 "PDET MunicipalPlanAnalyzer", "construct_causal_dag", context,  
4802:                 financial_analysis=financial_analysis  
4803:             )  
4804:             causal_effects = self._execute_method(  
4805:                 "PDET MunicipalPlanAnalyzer", "estimate_causal_effects", context,  
4806:                 dag=causal_dag,  
4807:                 financial_analysis=financial_analysis  
4808:             )  
4809:             counterfactuals = self._execute_method(  
4810:                 "PDET MunicipalPlanAnalyzer", "generate_counterfactuals", context,  
4811:                 dag=causal_dag,  
4812:                 causal_effects=causal_effects,  
4813:                 financial_analysis=financial_analysis
```

```
4814:         )
4815:         simulated_intervention = self._execute_method(
4816:             "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
4817:             intervention={},
4818:             dag=causal_dag,
4819:             causal_effects=causal_effects,
4820:             label="baseline"
4821:         )
4822:         causal_nodes = self._execute_method(
4823:             "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
4824:             text=context.get("document_text", ""),
4825:             tables=context.get("tables", []),
4826:             financial_analysis=financial_analysis
4827:         )
4828:         confounders = {}
4829:         for effect in causal_effects:
4830:             treatment = effect.treatment if hasattr(effect, "treatment") else None
4831:             outcome = effect.outcome if hasattr(effect, "outcome") else None
4832:             if treatment and outcome:
4833:                 confounders[(treatment, outcome)] = self._execute_method(
4834:                     "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
4835:                     treatment=treatment,
4836:                     outcome=outcome,
4837:                     dag=causal_dag
4838:                 )
4839:             effect_dicts = [
4840:                 self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context, effect=effect)
4841:                 for effect in causal_effects
4842:             ]
4843:             scenario_dicts = [
4844:                 self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict", context, scenario=scenario)
4845:                 for scenario in counterfactuals
4846:             ]
4847:             causal_recommendations = self._execute_method(
4848:                 "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
4849:                 analysis_results={"financial_analysis": financial_analysis, "quality_score": getattr(causal_dag, 'graph', {})}
4850:             )
4851:             factual_eval = None
4852:             counterfactual_eval = None
4853:             if causal_effects:
4854:                 first_effect = causal_effects[0]
4855:                 target = getattr(first_effect, "outcome", None) or ""
4856:                 evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
4857:                 factual_eval = self._execute_method(
4858:                     "BayesianCounterfactualAuditor", "_evaluate_factual", context,
4859:                     target=target,
4860:                     evidence=evidence
4861:                 )
4862:                 counterfactual_eval = self._execute_method(
4863:                     "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
4864:                     target=target,
4865:                     intervention={"shift": 0.1}
4866:                 )
4867:             # Only catch specific expected exceptions, let system exceptions propagate
4868:             matched_node = None
4869:             try:
```

```
4870:         matched_node = self._execute_method(
4871:             "PDET MunicipalPlanAnalyzer", "_match_text_to_node", context,
4872:             text=context.get("document_text", "")[:200],
4873:             nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
4874:         )
4875:     except (KeyError, ValueError, TypeError, AttributeError, ExecutorFailure) as e:
4876:         logger.warning(f"Node matching failed: {type(e).__name__}: {e}")
4877:         matched_node = None
4878:     # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError) propagate
4879:
4880:     # Step 1: Build type hierarchy
4881:     type_hierarchy = self._execute_method(
4882:         "CausalExtractor", "_build_type_hierarchy", context
4883:     )
4884:
4885:     # Step 2: Check structural violations
4886:     structuralViolations = self._execute_method(
4887:         "CausalExtractor", "_check_structuralViolation", context,
4888:         hierarchy=type_hierarchy
4889:     )
4890:
4891:     # Step 3: Calculate transition priors and proximity
4892:     transition_priors = self._execute_method(
4893:         "CausalExtractor", "_calculate_type_transition_prior", context,
4894:         hierarchy=type_hierarchy
4895:     )
4896:     textual_proximity = self._execute_method(
4897:         "CausalExtractor", "_calculate_textual_proximity", context
4898:     )
4899:
4900:     # Step 4: Validate causal order
4901:     causal_order_validation = self._execute_method(
4902:         "TeoriaCambio", "_validar_orden_causal", context,
4903:         hierarchy=type_hierarchy
4904:     )
4905:
4906:     # Step 5: Refine edge probabilities
4907:     refined_edges = self._execute_method(
4908:         "PDET MunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
4909:         priors=transition_priors
4910:     )
4911:     financial_consistency = None
4912:     if refined_edges:
4913:         first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
4914:         source = first_edge.get("source") if isinstance(first_edge, dict) else ""
4915:         target = first_edge.get("target") if isinstance(first_edge, dict) else ""
4916:         financial_consistency = self._execute_method(
4917:             "CausalExtractor", "_assess_financial_consistency", context,
4918:             source=source or "",
4919:             target=target or ""
4920:         )
4921:
4922:     # Step 6: Compare policy interventions
4923:     intervention_comparison = self._execute_method(
4924:         "PolicyAnalysisEmbedder", "compare_policy_interventions", context
4925:     )
```

```

4926:
4927:     # Step 7: Infer mechanisms
4928:     mechanisms = self._execute_method(
4929:         "BayesianMechanismInference", "infer_mechanisms", context,
4930:         edges=refined_edges
4931:     )
4932:     mechanism_sample = next(iter(mechanisms.values()), {})
4933:     activity_sequence = self._execute_method(
4934:         "BayesianMechanismInference", "_infer_activity_sequence", context,
4935:         observations=mechanism_sample.get("observations", {}),
4936:         mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0})
4937:     )
4938:     quantified_uncertainty = self._execute_method(
4939:         "BayesianMechanismInference", "_quantify_uncertainty", context,
4940:         mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0}),
4941:         sequence_posterior=mechanism_sample.get("activity_sequence", {}),
4942:         coherence_score=mechanism_sample.get("coherence_score", 0.0)
4943:     )
4944:     mechanism_observations = self._execute_method(
4945:         "BayesianMechanismInference", "_extract_observations", context,
4946:         node={"id": next(iter(mechanisms.keys()), "")},
4947:         text=context.get("document_text", "")
4948:     )
4949:     necessity_remediation = self._execute_method(
4950:         "BayesianMechanismInference", "_generate_necessity_remediation", context,
4951:         node_id=next(iter(mechanisms.keys()), ""),
4952:         missing_components=structural_violations
4953:     )
4954:     questionnaire_stub = self._execute_method(
4955:         "IndustrialPolicyProcessor", "_load_questionnaire", context
4956:     )
4957:     # Only catch specific expected exceptions, let system exceptions propagate
4958:     refutation_checks = None
4959:     try:
4960:         confounder_keys = list(confounders.keys())
4961:         first_pair = confounder_keys[0] if confounder_keys else ("", "")
4962:         refutation_checks = self._execute_method(
4963:             "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
4964:             dag=getattr(causal_dag, "graph", None),
4965:             target=first_pair[1],
4966:             treatment=first_pair[0],
4967:             confounders=list(confounders.values())[0] if confounders else []
4968:         )
4969:     except (KeyError, ValueError, TypeError, AttributeError, IndexError, ExecutorFailure) as e:
4970:         logger.warning(f"Refutation checks failed: {type(e).__name__}: {e}")
4971:         refutation_checks = None
4972:     # Let critical system exceptions (KeyboardInterrupt, SystemExit, MemoryError) propagate
4973:
4974:     raw_evidence = {
4975:         "output_outcome_links": refined_edges,
4976:         "mechanism_explanation": mechanisms,
4977:         "type_hierarchy": type_hierarchy,
4978:         "causal_dag": causal_dag,
4979:         "causal_effects": causal_effects,
4980:         "counterfactuals": counterfactuals,
4981:         "simulated_intervention": simulated_intervention,

```

```

4982:         "causal_nodes": causal_nodes,
4983:         "financial_analysis": financial_analysis,
4984:         "causal_validity": {
4985:             "structural_violations": structural_violations,
4986:             "order_valid": causal_order_validation
4987:         },
4988:         "transition_probabilities": transition_priors,
4989:         "textual_proximity": textual_proximity,
4990:         "intervention_comparison": intervention_comparison,
4991:         "confounders": confounders,
4992:         "effect_dicts": effect_dicts,
4993:         "scenario_dicts": scenario_dicts,
4994:         "activity_sequence_sample": activity_sequence,
4995:         "uncertainty_quantified": quantified_uncertainty,
4996:         "mechanism_observations": mechanism_observations,
4997:         "refutation_checks": refutation_checks,
4998:         "necessity_remediation": necessity_remediation,
4999:         "questionnaire_stub": questionnaire_stub,
5000:         "causal_recommendations": causal_recommendations,
5001:         "financial_consistency": financial_consistency,
5002:         "factual_eval": factual_eval,
5003:         "counterfactual_eval": counterfactual_eval,
5004:         "matched_node": matched_node
5005:     }
5006:
5007:     return {
5008:         "executor_id": self.executor_id,
5009:         "raw_evidence": raw_evidence,
5010:         "metadata": {
5011:             "methods_executed": [log["method"] for log in self.execution_log],
5012:             "mechanisms_identified": len(mechanisms or {}),
5013:             "violations_found": len(structural_violations or []),
5014:             "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
5015:             "dimension_code": dim_info.code,
5016:             "dimension_label": dim_info.label
5017:         },
5018:         "execution_metrics": {
5019:             "methods_count": len(self.execution_log),
5020:             "all_succeeded": all(log["success"] for log in self.execution_log)
5021:         }
5022:     }
5023:
5024:
5025: # =====
5026: # DIMENSION 4: RESULTS & OUTCOMES
5027: # =====
5028:
5029: class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
5030:     """
5031:     DIM04_Q01_OUTCOME_INDICATOR完整性 \u200\224 Validates outcome indicators (baseline, target, horizon) with canonical D4 notation.
5032:     Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical performance signals, and indicator quality scoring.
5033:
5034:     Methods (from D4-Q1):
5035:     - PDET Municipal Plan Analyzer._extract_entities_syntax
5036:     - PDET Municipal Plan Analyzer._extract_entities_ner
5037:     - CausalExtractor._calculate_language_specificity

```

```
5038:     - CausalExtractor._calculate_composite_likelihood
5039:     - CausalExtractor._calculate_semantic_distance
5040:     - TemporalLogicVerifier._classify_temporal_type
5041:     - PDET MunicipalPlanAnalyzer._score_indicators
5042:     - PDET MunicipalPlanAnalyzer._find_outcome_mentions
5043:     - PDET MunicipalPlanAnalyzer._score_temporal_consistency
5044:     - CausalExtractor._extract_goals
5045:     - CausalExtractor._parse_goal_context
5046:     - CausalExtractor._classify_goal_type
5047:     - TemporalLogicVerifier._parse_temporal_marker
5048:     - TemporalLogicVerifier._extract_resources
5049:     - TemporalLogicVerifier._should_precede
5050:     - PerformanceAnalyzer.analyze_performance
5051:     - PerformanceAnalyzer._generate_recommendations
5052: """
5053:
5054: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5055:     raw_evidence = {}
5056:     dim_info = get_dimension_info(CanonicalDimension.D4.value)
5057:
5058:     # Step 1: Find outcome mentions
5059:     outcome_mentions = self._execute_method(
5060:         "PDET MunicipalPlanAnalyzer", "_find_outcome_mentions", context
5061:     )
5062:     entities_syntax = self._execute_method(
5063:         "PDET MunicipalPlanAnalyzer", "_extract_entities_syntax", context,
5064:         text=context.get("document_text", "")
5065:     )
5066:     entities_syntax = self._safe_process_list(entities_syntax, label="entities_syntax")
5067:
5068:     entities_ner = self._execute_method(
5069:         "PDET MunicipalPlanAnalyzer", "_extract_entities_ner", context,
5070:         text=context.get("document_text", "")
5071:     )
5072:     entities_ner = self._safe_process_list(entities_ner, label="entities_ner")
5073:
5074:     # Step 2: Score temporal consistency
5075:     temporal_consistency = self._execute_method(
5076:         "PDET MunicipalPlanAnalyzer", "_score_temporal_consistency", context,
5077:         outcomes=outcome_mentions
5078:     )
5079:
5080:     # Step 3: Extract and classify goals
5081:     goals = self._execute_method(
5082:         "CausalExtractor", "_extract_goals", context
5083:     )
5084:     goal_contexts = self._execute_method(
5085:         "CausalExtractor", "_parse_goal_context", context,
5086:         goals=goals
5087:     )
5088:     goal_types = self._execute_method(
5089:         "CausalExtractor", "_classify_goal_type", context,
5090:         goals=goals
5091:     )
5092:     semantic_distance = 0.0
5093:     if goal_types and outcome_mentions:
```

```

5094:         semantic_distance = self._execute_method(
5095:             "CausalExtractor", "_calculate_semantic_distance", context,
5096:             source=str(goal_types[0]),
5097:             target=str(outcome_mentions[0])
5098:         )
5099:
5100:     # Step 4: Parse temporal markers
5101:     temporal_markers = self._execute_method(
5102:         "TemporalLogicVerifier", "_parse_temporal_marker", context,
5103:         contexts=goal_contexts
5104:     )
5105:     temporal_type = self._execute_method(
5106:         "TemporalLogicVerifier", "_classify_temporal_type", context,
5107:         marker=temporal_markers[0] if temporal_markers else ""
5108:     )
5109:     resources_mentioned = self._execute_method(
5110:         "TemporalLogicVerifier", "_extract_resources", context,
5111:         text=context.get("document_text", "")
5112:     )
5113:     precedence_check = self._execute_method(
5114:         "TemporalLogicVerifier", "_should_precede", context,
5115:         marker_a=temporal_markers[0] if temporal_markers else "",
5116:         marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
5117:     )
5118:
5119:     # Step 5: Analyze performance
5120:     performance_analysis = self._execute_method(
5121:         "PerformanceAnalyzer", "analyze_performance", context,
5122:         outcomes=outcome_mentions
5123:     )
5124:     indicator_quality = self._execute_method(
5125:         "PDET Municipal Plan Analyzer", "_score_indicators", context
5126:     )
5127:     performance_recommendations = self._execute_method(
5128:         "PerformanceAnalyzer", "_generate_recommendations", context,
5129:         performance_analysis=performance_analysis
5130:     )
5131:     # Semantic certainty for goals
5132:     language_specificity = self._execute_method(
5133:         "CausalExtractor", "_calculate_language_specificity", context,
5134:         keyword=goal_contexts[0] if goal_contexts else "",
5135:         policy_area=context.get("policy_area")
5136:     )
5137:     composite_likelihood = self._execute_method(
5138:         "CausalExtractor", "_calculate_composite_likelihood", context,
5139:         evidence={
5140:             "semantic_distance": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0,
5141:             "textual_proximity": performance_analysis.get("coherence_score", 0.0) if isinstance(performance_analysis, dict) else 0.0,
5142:             "language_specificity": language_specificity,
5143:             "temporal_coherence": temporal_consistency if isinstance(temporal_consistency, (int, float)) else 0.0
5144:         }
5145:     )
5146:
5147:     raw_evidence = {
5148:         "outcome_indicators": outcome_mentions,
5149:         "indicators_with_baseline": [o for o in outcome_mentions if o.get("has_baseline")],

```

```
5150:         "indicators_with_target": [o for o in outcome_mentions if o.get("has_target")],
5151:         "indicators_with_horizon": [o for o in outcome_mentions if o.get("time_horizon")],
5152:         "temporal_consistency_score": temporal_consistency,
5153:         "goal_classifications": goal_types,
5154:         "temporal_markers": temporal_markers,
5155:         "performance_metrics": performance_analysis,
5156:         "indicator_quality": indicator_quality,
5157:         "performance_recommendations": performance_recommendations,
5158:         "entities_syntax": entities_syntax,
5159:         "entities_ner": entities_ner,
5160:         "temporal_type": temporal_type,
5161:         "language_specificity": language_specificity,
5162:         "composite_likelihood": composite_likelihood,
5163:         "goal_outcome_semantic_distance": semantic_distance,
5164:         "resources_mentioned": resources_mentioned,
5165:         "precedence_check": precedence_check
5166:     }
5167:
5168:     memory_metrics = self._get_memory_metrics_summary()
5169:
5170:     return {
5171:         "executor_id": self.executor_id,
5172:         "raw_evidence": raw_evidence,
5173:         "metadata": {
5174:             "methods_executed": [log["method"] for log in self.execution_log],
5175:             "total_outcomes": len(outcome_mentions or []),
5176:             "complete_indicators": len([o for o in outcome_mentions or []
5177:                                         if o.get("has_baseline") and o.get("has_target") and o.get("time_horizon")]),
5178:             "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR完整性",
5179:             "dimension_code": dim_info.code,
5180:             "dimension_label": dim_info.label
5181:         },
5182:         "execution_metrics": {
5183:             "methods_count": len(self.execution_log),
5184:             "all_succeeded": all(log["success"] for log in self.execution_log),
5185:             "memory_safety": memory_metrics
5186:         }
5187:     }
5188:
5189:
5190: class D4_Q2_CausalChainValidator(BaseExecutor):
5191:     """
5192:     Validates explicit causal chain with assumptions and enabling conditions.
5193:
5194:     Methods (from D4-Q2):
5195:     - TeoriaCambio._encontrar_caminos_completos
5196:     - TeoriaCambio.validacion_completa
5197:     - CausalExtractor.extract_causal_hierarchy
5198:     - HierarchicalGenerativeModel.verify_conditional_independence
5199:     - HierarchicalGenerativeModel._generate_independence_tests
5200:     - BayesianCounterfactualAuditor.construct_scm
5201:     - AdvancedDAGValidator._perform_sensitivity_analysis_internal
5202:     - BayesFactorTable.get_bayes_factor
5203:     """
5204:
5205:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
```

```
5206:     raw_evidence = {}
5207:
5208:     # Step 1: Find complete causal paths
5209:     complete_paths = self._execute_method(
5210:         "TeoriaCambio", "_encontrar_caminos_completos", context
5211:     )
5212:
5213:     # Step 2: Complete validation
5214:     validation_results = self._execute_method(
5215:         "TeoriaCambio", "validacion_completa", context,
5216:         paths=complete_paths
5217:     )
5218:
5219:     # Step 3: Extract causal hierarchy
5220:     causal_hierarchy = self._execute_method(
5221:         "CausalExtractor", "extract_causal_hierarchy", context
5222:     )
5223:
5224:     # Step 4: Verify conditional independence
5225:     independence_verification = self._execute_method(
5226:         "HierarchicalGenerativeModel", "verify_conditional_independence", context,
5227:         hierarchy=causal_hierarchy
5228:     )
5229:     independence_tests = self._execute_method(
5230:         "HierarchicalGenerativeModel", "_generate_independence_tests", context,
5231:         verification=independence_verification
5232:     )
5233:
5234:     # Step 5: Construct structural causal model
5235:     scm = self._execute_method(
5236:         "BayesianCounterfactualAuditor", "construct_scm", context,
5237:         hierarchy=causal_hierarchy
5238:     )
5239:
5240:     # Step 6: Perform sensitivity analysis
5241:     sensitivity_analysis = self._execute_method(
5242:         "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
5243:         scm=scm
5244:     )
5245:
5246:     # Step 7: Get Bayes factor
5247:     bayes_factor = self._execute_method(
5248:         "BayesFactorTable", "get_bayes_factor", context,
5249:         analysis=sensitivity_analysis
5250:     )
5251:
5252:     raw_evidence = {
5253:         "causal_chain": complete_paths,
5254:         "key_assumptions": validation_results.get("assumptions", []),
5255:         "enabling_conditions": validation_results.get("conditions", []),
5256:         "external_factors": validation_results.get("external_factors", []),
5257:         "causal_hierarchy": causal_hierarchy,
5258:         "independence_tests": independence_tests,
5259:         "structural_model": scm,
5260:         "sensitivity": sensitivity_analysis,
5261:         "evidential_strength": bayes_factor
```

```
5262:         }
5263:
5264:     return {
5265:         "executor_id": self.executor_id,
5266:         "raw_evidence": raw_evidence,
5267:         "metadata": {
5268:             "methods_executed": [log["method"] for log in self.execution_log],
5269:             "complete_paths_found": len(complete_paths),
5270:             "assumptions_identified": len(validation_results.get("assumptions", []))
5271:         },
5272:         "execution_metrics": {
5273:             "methods_count": len(self.execution_log),
5274:             "all_succeeded": all(log["success"] for log in self.execution_log)
5275:         }
5276:     }
5277:
5278:
5279: class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
5280:     """
5281:     Analyzes justification of result ambition based on investment/capacity/benchmarks.
5282:
5283:     Methods (from D4-Q3):
5284:     - PDETMunicipalPlanAnalyzer._get_prior_effect
5285:     - PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
5286:     - PDETMunicipalPlanAnalyzer._compute_robustness_value
5287:     - AdaptivePriorCalculator.sensitivity_analysis
5288:     - HierarchicalGenerativeModel._calculate_r_hat
5289:     - HierarchicalGenerativeModel._calculate_ess
5290:     - AdvancedDAGValidator._calculate_statistical_power
5291:     - BayesianMechanismInference._aggregate_bayesian_confidence
5292:     """
5293:
5294:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5295:         raw_evidence = {}
5296:
5297:         # Step 1: Get prior effect estimates
5298:         prior_effects = self._execute_method(
5299:             "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
5300:         )
5301:
5302:         # Step 2: Estimate effect using Bayesian methods
5303:         effect_estimate = self._execute_method(
5304:             "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
5305:             priors=prior_effects
5306:         )
5307:
5308:         # Step 3: Compute robustness
5309:         robustness = self._execute_method(
5310:             "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
5311:             estimate=effect_estimate
5312:         )
5313:
5314:         # Step 4: Sensitivity analysis
5315:         sensitivity = self._execute_method(
5316:             "AdaptivePriorCalculator", "sensitivity_analysis", context,
5317:             estimate=effect_estimate
```

```
5318:     )
5319:
5320:     # Step 5: Calculate convergence diagnostics
5321:     r_hat = self._execute_method(
5322:         "HierarchicalGenerativeModel", "_calculate_r_hat", context
5323:     )
5324:     ess = self._execute_method(
5325:         "HierarchicalGenerativeModel", "_calculate_ess", context
5326:     )
5327:
5328:     # Step 6: Calculate statistical power
5329:     statistical_power = self._execute_method(
5330:         "AdvancedDAGValidator", "_calculate_statistical_power", context,
5331:         effect=effect_estimate
5332:     )
5333:
5334:     # Step 7: Aggregate confidence
5335:     confidence_aggregate = self._execute_method(
5336:         "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
5337:         estimates=[effect_estimate, robustness, statistical_power]
5338:     )
5339:
5340:     raw_evidence = {
5341:         "ambition_level": context.get("target_ambition", {}),
5342:         "financial_investment": context.get("total_investment", 0),
5343:         "institutional_capacity": context.get("capacity_score", 0),
5344:         "comparative_benchmarks": prior_effects,
5345:         "justification_analysis": {
5346:             "effect_estimate": effect_estimate,
5347:             "robustness": robustness,
5348:             "sensitivity": sensitivity,
5349:             "statistical_power": statistical_power
5350:         },
5351:         "convergence_diagnostics": {
5352:             "r_hat": r_hat,
5353:             "ess": ess
5354:         },
5355:         "overall_confidence": confidence_aggregate
5356:     }
5357:
5358:     return {
5359:         "executor_id": self.executor_id,
5360:         "raw_evidence": raw_evidence,
5361:         "metadata": {
5362:             "methods_executed": [log["method"] for log in self.execution_log],
5363:             "ambition_justified": confidence_aggregate > 0.7,
5364:             "statistical_power": statistical_power
5365:         },
5366:         "execution_metrics": {
5367:             "methods_count": len(self.execution_log),
5368:             "all_succeeded": all(log["success"] for log in self.execution_log)
5369:         }
5370:     }
5371:
5372:
5373: class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
```

```
5374: """
5375:     Evaluates whether results address/resolve prioritized problems from diagnosis.
5376:
5377:     Methods (from D4-Q4):
5378:         - PolicyContradictionDetector._calculate_objective_alignment
5379:         - PolicyContradictionDetector._identify_affected_sections
5380:         - PolicyContradictionDetector._generate_resolution_recommendations
5381:         - OperationalizationAuditor._generate_optimal_remediations
5382:         - OperationalizationAuditor._get_remediation_text
5383:         - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
5384:         - FinancialAuditor._detect_allocation_gaps
5385: """
5386:
5387: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5388:     raw_evidence = {}
5389:
5390:     # Step 1: Calculate objective alignment
5391:     objective_alignment = self._execute_method(
5392:         "PolicyContradictionDetector", "_calculate_objective_alignment", context
5393:     )
5394:
5395:     # Step 2: Identify affected sections
5396:     affected_sections = self._execute_method(
5397:         "PolicyContradictionDetector", "_identify_affected_sections", context,
5398:         alignment=objective_alignment
5399:     )
5400:
5401:     # Step 3: Generate resolution recommendations
5402:     resolutions = self._execute_method(
5403:         "PolicyContradictionDetector", "_generate_resolution_recommendations", context,
5404:         sections=affected_sections
5405:     )
5406:
5407:     # Step 4: Generate optimal remediations
5408:     remediations = self._execute_method(
5409:         "OperationalizationAuditor", "_generate_optimal_remediations", context
5410:     )
5411:     remediation_text = self._execute_method(
5412:         "OperationalizationAuditor", "_get_remediation_text", context,
5413:         remediations=remediations
5414:     )
5415:
5416:     # Step 5: Aggregate risk and prioritize
5417:     risk_priorities = self._execute_method(
5418:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
5419:     )
5420:
5421:     # Step 6: Detect allocation gaps
5422:     allocation_gaps = self._execute_method(
5423:         "FinancialAuditor", "_detect_allocation_gaps", context
5424:     )
5425:
5426:     raw_evidence = {
5427:         "prioritized_problems": context.get("diagnosis_problems", []),
5428:         "proposed_results": context.get("outcome_indicators", []),
5429:         "problem_result_mapping": objective_alignment,
```

```
5430:         "unaddressed_problems": [p for p in affected_sections if not p.get("addressed")],
5431:         "solvency_score": objective_alignment.get("score", 0),
5432:         "resolution_recommendations": resolutions,
5433:         "remediations": remediation_text,
5434:         "risk_priorities": risk_priorities,
5435:         "allocation_gaps": allocation_gaps
5436:     }
5437:
5438:     return {
5439:         "executor_id": self.executor_id,
5440:         "raw_evidence": raw_evidence,
5441:         "metadata": {
5442:             "methods_executed": [log["method"] for log in self.execution_log],
5443:             "problems_addressed": len([p for p in affected_sections if p.get("addressed")]),
5444:             "problems_unaddressed": len([p for p in affected_sections if not p.get("addressed")])
5445:         },
5446:         "execution_metrics": {
5447:             "methods_count": len(self.execution_log),
5448:             "all_succeeded": all(log["success"] for log in self.execution_log)
5449:         }
5450:     }
5451:
5452:
5453: class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
5454:     """
5455:     Validates alignment with superior frameworks (PND, SDGs).
5456:
5457:     Methods (from D4-Q5):
5458:     - PDETMunicipalPlanAnalyzer._score_pdet_alignment
5459:     - PDETMunicipalPlanAnalyzer._score_causal_coherence
5460:     - CDAFFramework._validate_dnp_compliance
5461:     - CDAFFramework._generate_dnp_report
5462:     - IndustrialPolicyProcessor._analyze_causal_dimensions
5463:     - AdaptivePriorCalculator.validate_quality_criteria
5464:     """
5465:
5466:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5467:         raw_evidence = {}
5468:
5469:         # Step 1: Score PDET alignment
5470:         pdet_alignment = self._execute_method(
5471:             "PDETMunicipalPlanAnalyzer", "_score_pdet_alignment", context
5472:         )
5473:
5474:         # Step 2: Score causal coherence
5475:         causal_coherence = self._execute_method(
5476:             "PDETMunicipalPlanAnalyzer", "_score_causal_coherence", context
5477:         )
5478:
5479:         # Step 3: Validate DNP compliance
5480:         dnp_compliance = self._execute_method(
5481:             "CDAFFramework", "_validate_dnp_compliance", context
5482:         )
5483:         dnp_report = self._execute_method(
5484:             "CDAFFramework", "_generate_dnp_report", context,
5485:             compliance=dnp_compliance
```

```
5486:     )
5487:
5488:     # Step 4: Analyze causal dimensions
5489:     causal_dimensions = self._execute_method(
5490:         "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
5491:     )
5492:
5493:     # Step 5: Validate quality criteria
5494:     quality_validation = self._execute_method(
5495:         "AdaptivePriorCalculator", "validate_quality_criteria", context,
5496:         alignment=pdet_alignment
5497:     )
5498:
5499:     raw_evidence = {
5500:         "pnd_alignment": dnp_compliance,
5501:         "sdg_alignment": context.get("sdg_mappings", []),
5502:         "pdet_alignment": pdet_alignment,
5503:         "alignment_declarations": (dnp_report or {}).get("declarations", []),
5504:         "causal_coherence": causal_coherence,
5505:         "causal_dimensions": causal_dimensions,
5506:         "quality_validation": quality_validation,
5507:         "alignment_score": (pdet_alignment.get("score", 0) +
5508:             (dnp_compliance or {}).get("score", 0)) / 2
5509:     }
5510:
5511:     return {
5512:         "executor_id": self.executor_id,
5513:         "raw_evidence": raw_evidence,
5514:         "metadata": {
5515:             "methods_executed": [log["method"] for log in self.execution_log],
5516:             "pnd_aligned": (dnp_compliance or {}).get("is_compliant", False),
5517:             "sdgs_referenced": len(context.get("sdg_mappings", []))
5518:         },
5519:         "execution_metrics": {
5520:             "methods_count": len(self.execution_log),
5521:             "all_succeeded": all(log["success"] for log in self.execution_log)
5522:         }
5523:     }
5524:
5525:
5526: # =====
5527: # DIMENSION 5: IMPACTS
5528: # =====
5529:
5530: class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
5531:     """
5532:         Analyzes long-term impacts, transmission routes, and time lags.
5533:
5534:         Methods (from D5-Q1):
5535:         - PDET Municipal Plan Analyzer generate_counterfactuals
5536:         - PDET Municipal Plan Analyzer._simulate_intervention
5537:         - PDET Municipal Plan Analyzer._generate_scenario_narrative
5538:         - PDET Municipal Plan Analyzer._find_mediator_mentions
5539:         - TeoriaCambio._validar_orden_causal
5540:         - CausalExtractor._assess_temporal_coherence
5541:         - TextMiningEngine._generate_interventions
```

```
5542:     - BayesianCounterfactualAuditor.construct_scm
5543:     """
5544:
5545:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5546:         raw_evidence = {}
5547:
5548:         # Step 1: Generate counterfactuals
5549:         counterfactuals = self._execute_method(
5550:             "PDET MunicipalPlanAnalyzer", "generate_counterfactuals", context
5551:         )
5552:
5553:         # Step 2: Simulate interventions
5554:         simulation = self._execute_method(
5555:             "PDET MunicipalPlanAnalyzer", "_simulate_intervention", context,
5556:             counterfactuals=counterfactuals
5557:         )
5558:
5559:         # Step 3: Generate scenario narratives
5560:         scenarios = self._execute_method(
5561:             "PDET MunicipalPlanAnalyzer", "_generate_scenario_narrative", context,
5562:             simulation=simulation
5563:         )
5564:
5565:         # Step 4: Find mediator mentions
5566:         mediators = self._execute_method(
5567:             "PDET MunicipalPlanAnalyzer", "_find_mediator_mentions", context
5568:         )
5569:
5570:         # Step 5: Validate causal order
5571:         causal_order = self._execute_method(
5572:             "TeoriaCambio", "_validar_orden_causal", context,
5573:             mediators=mediators
5574:         )
5575:
5576:         # Step 6: Assess temporal coherence
5577:         temporal_coherence = self._execute_method(
5578:             "CausalExtractor", "_assess_temporal_coherence", context
5579:         )
5580:
5581:         # Step 7: Generate interventions
5582:         interventions = self._execute_method(
5583:             "TextMiningEngine", "_generate_interventions", context
5584:         )
5585:
5586:         # Step 8: Construct SCM
5587:         scm = self._execute_method(
5588:             "BayesianCounterfactualAuditor", "construct_scm", context,
5589:             order=causal_order
5590:         )
5591:
5592:         raw_evidence = {
5593:             "long_term_impacts": context.get("impact_indicators", []),
5594:             "structural_transformations": scenarios,
5595:             "transmission_routes": mediators,
5596:             "expected_time_lags": temporal_coherence.get("time_lags", []),
5597:             "counterfactual_analysis": counterfactuals,
```

```
5598:         "simulation_results": simulation,
5599:         "causal_order_validation": causal_order,
5600:         "causal_pathways": scm,
5601:         "intervention_scenarios": interventions
5602:     }
5603:
5604:     return {
5605:         "executor_id": self.executor_id,
5606:         "raw_evidence": raw_evidence,
5607:         "metadata": {
5608:             "methods_executed": [log["method"] for log in self.execution_log],
5609:             "impacts_defined": len(context.get("impact_indicators", [])),
5610:             "mediators_identified": len(mediators or [])
5611:         },
5612:         "execution_metrics": {
5613:             "methods_count": len(self.execution_log),
5614:             "all_succeeded": all(log["success"] for log in self.execution_log)
5615:         }
5616:     }
5617:
5618:
5619: class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
5620:     """
5621:     DIM05_Q02_COMPOSITE_PROXY_VALIDITY \200\224 Validates composite indices/proxies for complex impacts (canonical D5).
5622:     Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative reporting quality, and semantic consistency.
5623:
5624:     Methods executed (in order):
5625:     Step 1: Quality score calculation - PDET Municipal Plan Analyzer.calculate_quality_score
5626:     Step 2: Score confidence estimation - PDET Municipal Plan Analyzer._estimate_score_confidence
5627:     Step 3: E-value computation - PDET Municipal Plan Analyzer._compute_e_value
5628:     Step 4: Robustness computation - PDET Municipal Plan Analyzer._compute_robustness_value
5629:     Step 5: Sensitivity interpretation - PDET Municipal Plan Analyzer._interpret_sensitivity
5630:     Step 6: Reporting quality - Reporting Engine._calculate_quality_score
5631:     Step 7: Bayesian confidence aggregation - Bayesian Mechanism Inference._aggregate_bayesian_confidence
5632:     Step 8: Numerical consistency evaluation - Policy Analysis Embedder.evaluate_policy_numerical_consistency
5633:     Step 9: Embedder diagnostics - Policy Analysis Embedder.get_diagnostics
5634:     Step 10: Document processing - Policy Analysis Embedder.process_document
5635:     Step 11: PDQ query generation - Policy Analysis Embedder._generate_query_from_pdq
5636:     Step 12: PDQ filtering - Policy Analysis Embedder._filter_by_pdq
5637:     Step 13: Numerical value extraction - Policy Analysis Embedder._extract_numerical_values
5638:     Step 14: Text embedding - Policy Analysis Embedder._embed_texts
5639:     Step 15: Overall confidence computation - Policy Analysis Embedder._compute_overall_confidence
5640:     Step 16: Sufficiency calculation - Financial Auditor._calculate_sufficiency
5641:     Step 17: Overall quality interpretation - PDET Municipal Plan Analyzer._interpret_overall_quality
5642:     Step 18: Risk prioritization - Bayesian Counterfactual Auditor.aggregate_risk_and_prioritize
5643:     Step 19: Unicode normalization - Policy Text Processor.normalize_unicode
5644:     Step 20: Sentence segmentation - Policy Text Processor.segment_into_sentences
5645:     Step 21: Evidence confidence - Industrial Policy Processor._compute_evidence_confidence
5646:     Step 22: Average confidence - Industrial Policy Processor._compute_avg_confidence
5647:     Step 23: Quality serialization - PDET Municipal Plan Analyzer._quality_to_dict
5648:     Step 24: Evidence bundle - Industrial Policy Processor._construct_evidence_bundle
5649:     Step 25: Pattern compilation - Policy Text Processor.compile_pattern
5650:     Step 26: Contextual window extraction - Policy Text Processor.extract_contextual_window
5651:     Step 27: Executive report generation - PDET Municipal Plan Analyzer.generate_executive_report
5652:     Step 28: Results export - Industrial Policy Processor.export_results
5653: """
```

```
5654:  
5655:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:  
5656:         raw_evidence = {}  
5657:         dim_info = get_dimension_info(CanonicalDimension.D5.value)  
5658:         document_text = context.get("document_text", "")  
5659:         document_metadata = context.get("metadata", {})  
5660:  
5661:         # Step 1: Calculate quality scores  
5662:         quality_score = self._execute_method(  
5663:             "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context  
5664:         )  
5665:         score_confidence = self._execute_method(  
5666:             "PDETMunicipalPlanAnalyzer", "estimate_score_confidence", context,  
5667:             score=quality_score  
5668:         )  
5669:  
5670:         # Step 2: Compute robustness metrics  
5671:         e_value = self._execute_method(  
5672:             "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,  
5673:             score=quality_score  
5674:         )  
5675:         robustness = self._execute_method(  
5676:             "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,  
5677:             score=quality_score  
5678:         )  
5679:         sensitivity_interpretation = self._execute_method(  
5680:             "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,  
5681:             e_value=e_value,  
5682:             robustness=robustness  
5683:         )  
5684:  
5685:         # Step 3: Calculate reporting quality score  
5686:         reporting_quality = self._execute_method(  
5687:             "ReportingEngine", "calculate_quality_score", context  
5688:         )  
5689:  
5690:         # Step 4: Aggregate Bayesian confidence  
5691:         bayesian_confidence = self._execute_method(  
5692:             "BayesianMechanismInference", "aggregate_bayesian_confidence", context,  
5693:             scores=[quality_score, reporting_quality]  
5694:         )  
5695:  
5696:         # Step 5: Evaluate numerical consistency  
5697:         numerical_consistency = self._execute_method(  
5698:             "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context  
5699:         )  
5700:         embedder_diagnostics = self._execute_method(  
5701:             "PolicyAnalysisEmbedder", "get_diagnostics", context  
5702:         )  
5703:         processed_chunks = self._execute_method(  
5704:             "PolicyAnalysisEmbedder", "process_document", context,  
5705:             document_text=document_text,  
5706:             document_metadata=document_metadata  
5707:         )  
5708:         processed_chunks = self._safe_process_list(processed_chunks, label="processed_chunks")  
5709:
```

```
5710:     pdq_filter = self._execute_method(
5711:         "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
5712:         pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
5713:     )
5714:     filtered_chunks = self._execute_method(
5715:         "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
5716:         chunks=processed_chunks,
5717:         pdq_filter=pdq_filter
5718:     )
5719:     filtered_chunks = self._safe_process_list(filtered_chunks, label="filtered_chunks")
5720:     numerical_values = self._execute_method(
5721:         "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
5722:         chunks=processed_chunks
5723:     )
5724:     embedded_texts = self._execute_method(
5725:         "PolicyAnalysisEmbedder", "_embed_texts", context,
5726:         texts=[c.get("content", "") for c in processed_chunks] if isinstance(processed_chunks, list) else []
5727:     )
5728:     overall_confidence = self._execute_method(
5729:         "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
5730:         relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else [],
5731:         numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict) else {"evidence_strength": "weak", "numerical_coherence": 0.0}
5732:     )
5733:
5734:     # Step 6: Calculate sufficiency
5735:     sufficiency = self._execute_method(
5736:         "FinancialAuditor", "_calculate_sufficiency", context
5737:     )
5738:     overall_interpretation = self._execute_method(
5739:         "PDET Municipal Plan Analyzer", "_interpret_overall_quality", context,
5740:         score=getattr(quality_score, "overall_score", quality_score)
5741:     )
5742:     risk_prioritization = self._execute_method(
5743:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
5744:         omission_score=1 - quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.2,
5745:         insufficiency_score=1 - (sufficiency or {}).get("coverage_ratio", 0.0),
5746:         unnecessary_score=1 - (robustness if isinstance(robustness, (int, float)) else 0.0),
5747:         causal_effect=e_value,
5748:         feasibility=quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.8,
5749:         cost=1.0
5750:     )
5751:     normalized_text = self._execute_method(
5752:         "PolicyTextProcessor", "normalize_unicode", context,
5753:         text=document_text
5754:     )
5755:     segmented_sentences = self._execute_method(
5756:         "PolicyTextProcessor", "segment_into_sentences", context,
5757:         text=document_text
5758:     )
5759:     evidence_confidence = self._execute_method(
5760:         "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
5761:         matches=context.get("proxy_indicators", []),
5762:         text_length=len(document_text),
5763:         pattern_specificity=0.5
5764:     )
5765:     avg_confidence = self._execute_method(
```

```
5766:         "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
5767:         dimension_analysis={"D5": {"dimension_confidence": (bayesian_confidence or {}).get("numerical_coherence", 0.0) if isinstance(bayesian_confidence
, dict) else 0.0)}
5768:     )
5769:     quality_dict = self._execute_method(
5770:         "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
5771:         quality=quality_score
5772:     )
5773:     evidence_bundle = self._execute_method(
5774:         "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
5775:         dimension=None,
5776:         category="composite",
5777:         matches=context.get("proxy_indicators", []),
5778:         positions=[],
5779:         confidence=bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0
5780:     )
5781:     compiled_pattern = self._execute_method(
5782:         "PolicyTextProcessor", "compile_pattern", context,
5783:         pattern_str=r"[A-Z]{2,}\s+\d+"
5784:     )
5785:     contextual_window = self._execute_method(
5786:         "PolicyTextProcessor", "extract_contextual_window", context,
5787:         text=document_text,
5788:         match_position=0,
5789:         window_size=200
5790:     )
5791:     exec_report = self._execute_method(
5792:         "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
5793:         analysis_results={"quality_score": quality_dict, "financial_analysis": context.get("financial_analysis", {}) or {"total_budget": 0, "funding_sou
rces": {}, "confidence": (0, 0)}}
5794:     )
5795:     export_result = self._execute_method(
5796:         "IndustrialPolicyProcessor", "export_results", context,
5797:         results={"quality": quality_dict, "robustness": robustness},
5798:         output_path="output/composite_results.json"
5799:     )
5800:
5801:     raw_evidence = {
5802:         "composite_indices": context.get("composite_indicators", []),
5803:         "proxy_indicators": context.get("proxy_indicators", []),
5804:         "validity_justification": score_confidence,
5805:         "robustness_metrics": {
5806:             "e_value": e_value,
5807:             "robustness": robustness,
5808:             "interpretation": sensitivity_interpretation
5809:         },
5810:         "quality_scores": {
5811:             "overall": quality_score,
5812:             "reporting": reporting_quality
5813:         },
5814:         "bayesian_confidence": bayesian_confidence,
5815:         "numerical_consistency": numerical_consistency,
5816:         "measurement_sufficiency": sufficiency,
5817:         "embedder_diagnostics": embedder_diagnostics,
5818:         "quality_interpretation": overall_interpretation,
5819:         "pdq_filter": pdq_filter,
```

```
5820:         "filtered_chunks": filtered_chunks,
5821:         "numerical_values": numerical_values,
5822:         "embedded_texts": embedded_texts,
5823:         "overall_confidence": overall_confidence,
5824:         "risk_prioritization": risk_prioritization,
5825:         "normalized_text": normalized_text,
5826:         "segmented_sentences": segmented_sentences,
5827:         "evidence_confidence": evidence_confidence,
5828:         "avg_confidence": avg_confidence,
5829:         "quality_dict": quality_dict,
5830:         "compiled_pattern": compiled_pattern,
5831:         "contextual_window": contextual_window,
5832:         "evidence_bundle": evidence_bundle,
5833:         "executive_report": exec_report,
5834:         "export_result": export_result
5835:     }
5836:
5837:     return {
5838:         "executor_id": self.executor_id,
5839:         "raw_evidence": raw_evidence,
5840:         "metadata": {
5841:             "methods_executed": [log["method"] for log in self.execution_log],
5842:             "composite_indices_count": len(context.get("composite_indicators", [])),
5843:             "total_proxy_indicators": len(context.get("proxy_indicators", [])),
5844:             "has_proxy_indicators": bool(context.get("proxy_indicators")),
5845:             "total_numerical_values": len(numerical_values) if numerical_values else 0,
5846:             "has_numerical_values": bool(numerical_values),
5847:             "total_filtered_chunks": len(filtered_chunks) if filtered_chunks else 0,
5848:             "has_filtered_chunks": bool(filtered_chunks),
5849:             "total_segmented_sentences": len(segmented_sentences) if segmented_sentences else 0,
5850:             "has_segmented_sentences": bool(segmented_sentences),
5851:             "total_embedded_texts": len(embedded_texts) if embedded_texts else 0,
5852:             "has_embedded_texts": bool(embedded_texts),
5853:             "validity_score": score_confidence,
5854:             "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
5855:             "dimension_code": dim_info.code,
5856:             "dimension_label": dim_info.label
5857:         },
5858:         "execution_metrics": {
5859:             "methods_count": len(self.execution_log),
5860:             "all_succeeded": all(log["success"] for log in self.execution_log),
5861:             "memory_safety": self._get_memory_metrics_summary()
5862:         }
5863:     }
5864:
5865:
5866: class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
5867:     """
5868:     Analyzes proxy indicators for intangible impacts with validity documentation.
5869:
5870:     Methods (from D5-Q3):
5871:     - CausalExtractor._calculate_semantic_distance
5872:     - SemanticAnalyzer.extract_semantic_cube
5873:     - BayesianMechanismInference._quantify_uncertainty
5874:     - PDET MunicipalPlanAnalyzer._find_mediator_mentions
5875:     - PolicyAnalysisEmbedder.get_diagnostics
```

```
5876:     - AdaptivePriorCalculator._perturb_evidence
5877:     """
5878:
5879:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5880:         raw_evidence = {}
5881:
5882:         # Step 1: Calculate semantic distance
5883:         semantic_distance = self._execute_method(
5884:             "CausalExtractor", "_calculate_semantic_distance", context
5885:         )
5886:
5887:         # Step 2: Extract semantic cube
5888:         semantic_cube = self._execute_method(
5889:             "SemanticAnalyzer", "extract_semantic_cube", context
5890:         )
5891:
5892:         # Step 3: Quantify uncertainty
5893:         uncertainty = self._execute_method(
5894:             "BayesianMechanismInference", "_quantify_uncertainty", context,
5895:             semantic_data=semantic_cube
5896:         )
5897:
5898:         # Step 4: Find mediator mentions
5899:         mediators = self._execute_method(
5900:             "PDET MunicipalPlanAnalyzer", "_find_mediator_mentions", context
5901:         )
5902:
5903:         # Step 5: Get diagnostics
5904:         diagnostics = self._execute_method(
5905:             "PolicyAnalysisEmbedder", "get_diagnostics", context,
5906:             mediators=mediators
5907:         )
5908:
5909:         # Step 6: Perturb evidence for sensitivity
5910:         perturbed_evidence = self._execute_method(
5911:             "AdaptivePriorCalculator", "_perturb_evidence", context,
5912:             diagnostics=diagnostics
5913:         )
5914:
5915:         raw_evidence = {
5916:             "intangible_impacts": context.get("intangible_indicators", []),
5917:             "proxy_indicators": context.get("proxy_mappings", []),
5918:             "validity_documentation": diagnostics,
5919:             "limitations_acknowledged": (diagnostics or {}).get("limitations", []),
5920:             "semantic_relationships": semantic_cube,
5921:             "semantic_distance": semantic_distance,
5922:             "uncertainty_quantification": uncertainty,
5923:             "sensitivity_to_proxies": perturbed_evidence
5924:         }
5925:
5926:         return {
5927:             "executor_id": self.executor_id,
5928:             "raw_evidence": raw_evidence,
5929:             "metadata": {
5930:                 "methods_executed": [log["method"] for log in self.execution_log],
5931:                 "intangibles_count": len(context.get("intangible_indicators", [])),
5932:             }
5933:         }
```

```
5932:         "proxies_defined": len(context.get("proxy_mappings", []))
5933:     },
5934:     "execution_metrics": {
5935:         "methods_count": len(self.execution_log),
5936:         "all_succeeded": all(log["success"] for log in self.execution_log)
5937:     }
5938: }
5939:
5940:
5941: class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
5942:     """
5943:     Evaluates systemic risks that could rupture causal mechanisms.
5944:
5945:     Methods (from D5-Q4):
5946:     - OperationalizationAuditor._audit_systemic_risk
5947:     - BayesianCounterfactualAuditor.refutation_and_sanity_checks
5948:     - BayesianCounterfactualAuditor._test_effect_stability
5949:     - PDET MunicipalPlanAnalyzer._interpret_risk
5950:     - PDET MunicipalPlanAnalyzer._interpret_sensitivity
5951:     - PDET MunicipalPlanAnalyzer._break_cycles
5952:     - AdaptivePriorCalculator.sensitivity_analysis
5953:     """
5954:
5955:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
5956:         raw_evidence = {}
5957:
5958:         # Step 1: Audit systemic risks
5959:         systemic_risks = self._execute_method(
5960:             "OperationalizationAuditor", "_audit_systemic_risk", context
5961:         )
5962:
5963:         # Step 2: Refutation and sanity checks
5964:         refutation = self._execute_method(
5965:             "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
5966:             risks=systemic_risks
5967:         )
5968:
5969:         # Step 3: Test effect stability
5970:         effect_stability = self._execute_method(
5971:             "BayesianCounterfactualAuditor", "_test_effect_stability", context,
5972:             refutation=refutation
5973:         )
5974:
5975:         # Step 4: Interpret risks
5976:         risk_interpretation = self._execute_method(
5977:             "PDET MunicipalPlanAnalyzer", "_interpret_risk", context,
5978:             risks=systemic_risks
5979:         )
5980:
5981:         # Step 5: Interpret sensitivity
5982:         sensitivity_interpretation = self._execute_method(
5983:             "PDET MunicipalPlanAnalyzer", "_interpret_sensitivity", context,
5984:             stability=effect_stability
5985:         )
5986:
5987:         # Step 6: Break cycles if present
```

```

5988:         cycle_breaks = self._execute_method(
5989:             "PDET MunicipalPlanAnalyzer", "_break_cycles", context
5990:         )
5991:
5992:         # Step 7: Sensitivity analysis
5993:         sensitivity = self._execute_method(
5994:             "AdaptivePriorCalculator", "sensitivity_analysis", context,
5995:             risks=systemic_risks
5996:         )
5997:
5998:         raw_evidence = {
5999:             "macroeconomic_risks": [r for r in systemic_risks or [] if r.get("type") == "macroeconomic"],
6000:             "environmental_risks": [r for r in systemic_risks or [] if r.get("type") == "environmental"],
6001:             "political_risks": [r for r in systemic_risks or [] if r.get("type") == "political"],
6002:             "mechanism_rupture_potential": (risk_interpretation or {}).get("rupture_probability", 0),
6003:             "effect_stability": effect_stability,
6004:             "refutation_results": refutation,
6005:             "sensitivity_analysis": sensitivity,
6006:             "sensitivity_interpretation": sensitivity_interpretation,
6007:             "cycle_vulnerabilities": cycle_breaks
6008:         }
6009:
6010:     return {
6011:         "executor_id": self.executor_id,
6012:         "raw_evidence": raw_evidence,
6013:         "metadata": {
6014:             "methods_executed": [log["method"] for log in self.execution_log],
6015:             "systemic_risks_identified": len(systemic_risks or []),
6016:             "high_risk_count": len([r for r in systemic_risks or [] if r.get("severity") == "high"])
6017:         },
6018:         "execution_metrics": {
6019:             "methods_count": len(self.execution_log),
6020:             "all_succeeded": all(log["success"] for log in self.execution_log)
6021:         }
6022:     }
6023:
6024:
6025: class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
6026:     """
6027:         Analyzes realism of impact ambition and potential unintended effects.
6028:
6029:         Methods (from D5-Q5):
6030:         - HierarchicalGenerativeModel.posterior_predictive_check
6031:         - HierarchicalGenerativeModel._ablation_analysis
6032:         - HierarchicalGenerativeModel._calculate_waic_difference
6033:         - AdaptivePriorCalculator._add_ood_noise
6034:         - AdaptivePriorCalculator.validate_quality_criteria
6035:         - PDET MunicipalPlanAnalyzer._compute_e_value
6036:         - PDET MunicipalPlanAnalyzer._compute_robustness_value
6037:         - BayesianMechanismInference._calculate_coherence_factor
6038:     """
6039:
6040:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6041:         raw_evidence = {}
6042:
6043:         # Step 1: Posterior predictive check

```

```
6044:     predictive_check = self._execute_method(
6045:         "HierarchicalGenerativeModel", "posterior_predictive_check", context
6046:     )
6047:
6048:     # Step 2: Ablation analysis
6049:     ablation = self._execute_method(
6050:         "HierarchicalGenerativeModel", "_ablation_analysis", context,
6051:         check=predictive_check
6052:     )
6053:
6054:     # Step 3: Calculate WAIC difference
6055:     waic_diff = self._execute_method(
6056:         "HierarchicalGenerativeModel", "_calculate_waic_difference", context,
6057:         ablation=ablation
6058:     )
6059:
6060:     # Step 4: Add out-of-distribution noise
6061:     ood_analysis = self._execute_method(
6062:         "AdaptivePriorCalculator", "_add_ood_noise", context
6063:     )
6064:
6065:     # Step 5: Validate quality criteria
6066:     quality_validation = self._execute_method(
6067:         "AdaptivePriorCalculator", "validate_quality_criteria", context,
6068:         ood=ood_analysis
6069:     )
6070:
6071:     # Step 6: Compute robustness metrics
6072:     e_value = self._execute_method(
6073:         "PDET MunicipalPlanAnalyzer", "_compute_e_value", context
6074:     )
6075:     robustness = self._execute_method(
6076:         "PDET MunicipalPlanAnalyzer", "_compute_robustness_value", context
6077:     )
6078:
6079:     # Step 7: Calculate coherence factor
6080:     coherence = self._execute_method(
6081:         "BayesianMechanismInference", "_calculate_coherence_factor", context
6082:     )
6083:
6084:     raw_evidence = {
6085:         "impact_ambition_level": context.get("declared_ambition", 0),
6086:         "realism_assessment": (predictive_check or {}).get("realism_score", 0),
6087:         "negative_side_effects": (ablation or {}).get("negative_effects", []),
6088:         "limit_hypotheses": (quality_validation or {}).get("limits", []),
6089:         "robustness_metrics": {
6090:             "e_value": e_value,
6091:             "robustness": robustness,
6092:             "coherence": coherence
6093:         },
6094:         "predictive_validity": predictive_check,
6095:         "ablation_results": ablation,
6096:         "model_comparison": waic_diff,
6097:         "ood_sensitivity": ood_analysis
6098:     }
6099:
```

```
6100:         return {
6101:             "executor_id": self.executor_id,
6102:             "raw_evidence": raw_evidence,
6103:             "metadata": {
6104:                 "methods_executed": [log["method"] for log in self.execution_log],
6105:                 "realism_score": (predictive_check or {}).get("realism_score", 0),
6106:                 "side_effects_identified": len((ablation or {}).get("negative_effects", []))
6107:             },
6108:             "execution_metrics": {
6109:                 "methods_count": len(self.execution_log),
6110:                 "all_succeeded": all(log["success"] for log in self.execution_log)
6111:             }
6112:         }
6113:
6114:
6115: # =====
6116: # DIMENSION 6: CAUSALITY & THEORY OF CHANGE
6117: # =====
6118:
6119: class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
6120:     """
6121:     Builds/validates explicit Theory of Change with diagram and assumptions.
6122:
6123:     Methods (from D6-Q1):
6124:     - TeoriaCambio.construir_grafo_causal
6125:     - TeoriaCambio.validacion_completa
6126:     - TeoriaCambio.export_nodes
6127:     - ReportingEngine.generate_causal_diagram
6128:     - ReportingEngine.generate_causal_model_json
6129:     - AdvancedDAGValidator.export_nodes
6130:     - PDET MunicipalPlanAnalyzer.export_causal_network
6131:     - CausalExtractor.extract_causal_hierarchy
6132:     """
6133:
6134:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6135:         raw_evidence = {}
6136:
6137:         # Step 1: Build causal graph
6138:         causal_graph = self._execute_method(
6139:             "TeoriaCambio", "construir_grafo_causal", context
6140:         )
6141:
6142:         # Step 2: Complete validation
6143:         validation = self._execute_method(
6144:             "TeoriaCambio", "validacion_completa", context,
6145:             graph=causal_graph
6146:         )
6147:
6148:         causal_graph = self._safe_process_dict(causal_graph, label="causal_graph")
6149:
6150:         # Step 3: Export nodes from Theory of Change
6151:         toc_nodes = self._execute_method(
6152:             "TeoriaCambio", "export_nodes", context,
6153:             graph=causal_graph
6154:         )
6155:         toc_nodes = self._safe_process_list(toc_nodes, label="toc_nodes")
```

```
6156:  
6157:      # Step 4: Generate causal diagram  
6158:      diagram = self._execute_method(  
6159:          "ReportingEngine", "generate_causal_diagram", context,  
6160:          graph=causal_graph  
6161:      )  
6162:  
6163:      # Step 5: Generate causal model JSON  
6164:      model_json = self._execute_method(  
6165:          "ReportingEngine", "generate_causal_model_json", context,  
6166:          graph=causal_graph  
6167:      )  
6168:      model_json = self._safe_process_dict(model_json, label="causal_model_json")  
6169:  
6170:      # Step 6: Export nodes from DAG validator  
6171:      dag_nodes = self._execute_method(  
6172:          "AdvancedDAGValidator", "export_nodes", context,  
6173:          graph=causal_graph  
6174:      )  
6175:      dag_nodes = self._safe_process_list(dag_nodes, label="dag_nodes")  
6176:  
6177:      # Step 7: Export causal network  
6178:      network_export = self._execute_method(  
6179:          "PDETMunicipalPlanAnalyzer", "export_causal_network", context,  
6180:          graph=causal_graph  
6181:      )  
6182:      network_export = self._safe_process_dict(network_export, label="network_export")  
6183:  
6184:      # Step 8: Extract causal hierarchy  
6185:      hierarchy = self._execute_method(  
6186:          "CausalExtractor", "extract_causal_hierarchy", context  
6187:      )  
6188:      hierarchy = self._safe_process_dict(hierarchy, label="causal_hierarchy")  
6189:  
6190:      raw_evidence = {  
6191:          "toc_exists": bool(causal_graph),  
6192:          "toc_diagram": diagram,  
6193:          "toc_json": model_json,  
6194:          "causal_graph": causal_graph,  
6195:          "nodes": toc_nodes,  
6196:          "dag_nodes": dag_nodes,  
6197:          "causes_identified": (hierarchy or {}).get("causes", []),  
6198:          "mediators_identified": (hierarchy or {}).get("mediators", []),  
6199:          "assumptions": (validation or {}).get("assumptions", []),  
6200:          "network_structure": network_export,  
6201:          "validation_results": validation  
6202:      }  
6203:  
6204:      memory_metrics = self._get_memory_metrics_summary()  
6205:  
6206:      return {  
6207:          "executor_id": self.executor_id,  
6208:          "raw_evidence": raw_evidence,  
6209:          "metadata": {  
6210:              "methods_executed": [log["method"] for log in self.execution_log],  
6211:              "nodes_count": len(toc_nodes or []),  
6212:          },  
6213:      }
```

```
6212:         "assumptions_count": len((validation or {}).get("assumptions", []))
6213:     },
6214:     "execution_metrics": {
6215:         "methods_count": len(self.execution_log),
6216:         "all_succeeded": all(log["success"] for log in self.execution_log),
6217:         "memory_safety": memory_metrics
6218:     }
6219: }
6220:
6221:
6222: class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
6223: """
6224:     Validates logical proportionality: no leaps, intervention matches result scale.
6225:
6226:     Methods (from D6-Q2):
6227:     - BeachEvidentialTest.apply_test_logic
6228:     - BayesianMechanismInference._test_necessity
6229:     - BayesianMechanismInference._test_sufficiency
6230:     - BayesianMechanismInference._calculate_coherence_factor
6231:     - BayesianCounterfactualAuditor._test_effect_stability
6232:     - IndustrialGradeValidator.validate_connection_matrix
6233:     - PolicyAnalysisEmbedder._compute_overall_confidence
6234: """
6235:
6236: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6237:     raw_evidence = {}
6238:
6239:     # Step 1: Apply evidential tests
6240:     evidential_tests = self._execute_method(
6241:         "BeachEvidentialTest", "apply_test_logic", context
6242:     )
6243:
6244:     # Step 2: Test necessity
6245:     necessity_test = self._execute_method(
6246:         "BayesianMechanismInference", "_test_necessity", context
6247:     )
6248:
6249:     # Step 3: Test sufficiency
6250:     sufficiency_test = self._execute_method(
6251:         "BayesianMechanismInference", "_test_sufficiency", context
6252:     )
6253:
6254:     # Step 4: Calculate coherence factor
6255:     coherence_factor = self._execute_method(
6256:         "BayesianMechanismInference", "_calculate_coherence_factor", context,
6257:         necessity=necessity_test,
6258:         sufficiency=sufficiency_test
6259:     )
6260:
6261:     # Step 5: Test effect stability
6262:     effect_stability = self._execute_method(
6263:         "BayesianCounterfactualAuditor", "_test_effect_stability", context
6264:     )
6265:
6266:     # Step 6: Validate connection matrix
6267:     connection_validation = self._execute_method(
```

```

6268:         "IndustrialGradeValidator", "validate_connection_matrix", context
6269:     )
6270:
6271:     # Step 7: Compute overall confidence
6272:     overall_confidence = self._execute_method(
6273:         "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
6274:         tests=[necessity_test, sufficiency_test, effect_stability]
6275:     )
6276:
6277:     raw_evidence = {
6278:         "logical_leaps_detected": (evidential_tests or {}).get("leaps", []),
6279:         "intervention_scale": context.get("intervention_magnitude", 0),
6280:         "result_scale": context.get("result_magnitude", 0),
6281:         "proportionality_ratio": context.get("intervention_magnitude", 0) / max(context.get("result_magnitude", 1), 1),
6282:         "necessity_score": necessity_test,
6283:         "sufficiency_score": sufficiency_test,
6284:         "coherence_factor": coherence_factor,
6285:         "effect_stability": effect_stability,
6286:         "connection_validation": connection_validation,
6287:         "overall_confidence": overall_confidence,
6288:         "implementation_miracles": [leap for leap in (evidential_tests or {}).get("leaps", [])
6289:                                         if isinstance(leap, dict) and leap.get("type") == "miracle"]
6290:     }
6291:
6292:     return {
6293:         "executor_id": self.executor_id,
6294:         "raw_evidence": raw_evidence,
6295:         "metadata": {
6296:             "methods_executed": [log["method"] for log in self.execution_log],
6297:             "leaps_detected": len((evidential_tests or {}).get("leaps", [])),
6298:             "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] - 1.0) < 0.5
6299:         },
6300:         "execution_metrics": {
6301:             "methods_count": len(self.execution_log),
6302:             "all_succeeded": all(log["success"] for log in self.execution_log)
6303:         }
6304:     }
6305:
6306:
6307: class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
6308:     """
6309:     Analyzes validation/testing proposals for weak assumptions before scaling.
6310:
6311:     Methods (from D6-Q3):
6312:     - IndustrialGradeValidator.execute_suite
6313:     - IndustrialGradeValidator.validate_engine_readiness
6314:     - IndustrialGradeValidator._benchmark_operation
6315:     - AdaptivePriorCalculator.validate_quality_criteria
6316:     - HierarchicalGenerativeModel._calculate_r_hat
6317:     - HierarchicalGenerativeModel._calculate_ess
6318:     - AdvancedDAGValidator.calculateACYCLICITY_pvalue
6319:     - PerformanceAnalyzer.analyze_performance
6320:     """
6321:
6322:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6323:         raw_evidence = {}

```

```
6324:  
6325:     # Step 1: Execute validation suite  
6326:     validation_suite = self._execute_method(  
6327:         "IndustrialGradeValidator", "execute_suite", context  
6328:     )  
6329:  
6330:     # Step 2: Validate engine readiness  
6331:     readiness = self._execute_method(  
6332:         "IndustrialGradeValidator", "validate_engine_readiness", context  
6333:     )  
6334:  
6335:     # Step 3: Benchmark operations  
6336:     benchmarks = self._execute_method(  
6337:         "IndustrialGradeValidator", "_benchmark_operation", context  
6338:     )  
6339:  
6340:     # Step 4: Validate quality criteria  
6341:     quality_validation = self._execute_method(  
6342:         "AdaptivePriorCalculator", "validate_quality_criteria", context  
6343:     )  
6344:  
6345:     # Step 5: Calculate convergence diagnostics  
6346:     r_hat = self._execute_method(  
6347:         "HierarchicalGenerativeModel", "_calculate_r_hat", context  
6348:     )  
6349:     ess = self._execute_method(  
6350:         "HierarchicalGenerativeModel", "_calculate_ess", context  
6351:     )  
6352:  
6353:     # Step 6: Calculate acyclicity p-value  
6354:     acyclicity_p = self._execute_method(  
6355:         "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context  
6356:     )  
6357:  
6358:     # Step 7: Analyze performance  
6359:     performance = self._execute_method(  
6360:         "PerformanceAnalyzer", "analyze_performance", context  
6361:     )  
6362:  
6363:     raw_evidence = {  
6364:         "inconsistencies_recognized": (validation_suite or {}).get("inconsistencies", []),  
6365:         "weak_assumptions": (quality_validation or {}).get("weak_assumptions", []),  
6366:         "pilot_proposals": context.get("pilot_programs", []),  
6367:         "testing_proposals": context.get("testing_plans", []),  
6368:         "validation_before_scaling": (readiness or {}).get("ready_to_scale", False),  
6369:         "validation_results": validation_suite,  
6370:         "quality_criteria": quality_validation,  
6371:         "convergence_diagnostics": {  
6372:             "r_hat": r_hat,  
6373:             "ess": ess,  
6374:             "acyclicity_p": acyclicity_p  
6375:         },  
6376:         "performance_analysis": performance,  
6377:         "benchmarks": benchmarks  
6378:     }  
6379:
```

```
6380:         return {
6381:             "executor_id": self.executor_id,
6382:             "raw_evidence": raw_evidence,
6383:             "metadata": {
6384:                 "methods_executed": [log["method"] for log in self.execution_log],
6385:                 "inconsistencies_count": len((validation_suite or {}).get("inconsistencies", [])),
6386:                 "pilots_proposed": len(context.get("pilot_programs", []))
6387:             },
6388:             "execution_metrics": {
6389:                 "methods_count": len(self.execution_log),
6390:                 "all_succeeded": all(log["success"] for log in self.execution_log)
6391:             }
6392:         }
6393:
6394:
6395: class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):
6396:     """
6397:         Analyzes monitoring system with correction mechanisms and learning processes.
6398:
6399:     Methods (from D6-Q4):
6400:     - ConfigLoader.update_priors_from_feedback
6401:     - ConfigLoader.check_uncertainty_reduction_criterion
6402:     - ConfigLoader._save_prior_history
6403:     - ConfigLoader._load_uncertainty_history
6404:     - CDAFFramework._extract_feedback_from_audit
6405:     - AdvancedDAGValidator._calculate_node_importance
6406:     - BayesFactorTable.get_bayes_factor
6407:     """
6408:
6409:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6410:         raw_evidence = {}
6411:
6412:         # Step 1: Update priors from feedback
6413:         prior_updates = self._execute_method(
6414:             "ConfigLoader", "update_priors_from_feedback", context
6415:         )
6416:
6417:         # Step 2: Check uncertainty reduction
6418:         uncertainty_reduction = self._execute_method(
6419:             "ConfigLoader", "check_uncertainty_reduction_criterion", context,
6420:             updates=prior_updates
6421:         )
6422:
6423:         # Step 3: Save prior history
6424:         history_saved = self._execute_method(
6425:             "ConfigLoader", "_save_prior_history", context,
6426:             updates=prior_updates
6427:         )
6428:
6429:         # Step 4: Load uncertainty history
6430:         uncertainty_history = self._execute_method(
6431:             "ConfigLoader", "_load_uncertainty_history", context
6432:         )
6433:
6434:         # Step 5: Extract feedback from audit
6435:         feedback_extracted = self._execute_method(
```

```

6436:         "CDAFFramework", "_extract_feedback_from_audit", context
6437:     )
6438:
6439:     # Step 6: Calculate node importance
6440:     node_importance = self._execute_method(
6441:         "AdvancedDAGValidator", "_calculate_node_importance", context
6442:     )
6443:
6444:     # Step 7: Get Bayes factor
6445:     bayes_factor = self._execute_method(
6446:         "BayesFactorTable", "get_bayes_factor", context,
6447:         updates=prior_updates
6448:     )
6449:
6450:     raw_evidence = {
6451:         "monitoring_system_described": len(context.get("monitoring_indicators", [])) > 0,
6452:         "correction_mechanisms": (feedback_extracted or {}).get("mechanisms", []),
6453:         "feedback_loops": (feedback_extracted or {}).get("loops", []),
6454:         "learning_processes": (feedback_extracted or {}).get("learning", []),
6455:         "prior_updates": prior_updates,
6456:         "uncertainty_reduction": uncertainty_reduction,
6457:         "history_saved": history_saved,
6458:         "uncertainty_history": uncertainty_history,
6459:         "node_importance": node_importance,
6460:         "learning_strength": bayes_factor
6461:     }
6462:
6463:     return {
6464:         "executor_id": self.executor_id,
6465:         "raw_evidence": raw_evidence,
6466:         "metadata": {
6467:             "methods_executed": [log["method"] for log in self.execution_log],
6468:             "feedback_mechanisms": len((feedback_extracted or {}).get("mechanisms", [])),
6469:             "learning_processes": len((feedback_extracted or {}).get("learning", []))
6470:         },
6471:         "execution_metrics": {
6472:             "methods_count": len(self.execution_log),
6473:             "all_succeeded": all(log["success"] for log in self.execution_log)
6474:         }
6475:     }
6476:
6477:
6478: class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
6479: """
6480:     Evaluates contextual adaptation: differential impacts and territorial constraints.
6481:
6482:     Methods executed (in order):
6483:         Step 1: Language specificity - CausalExtractor._calculate_language_specificity
6484:         Step 2: Temporal coherence - CausalExtractor._assess_temporal_coherence
6485:         Step 3: Critical links diagnosis - TextMiningEngine.diagnose_critical_links
6486:         Step 4: Failure points identification - CausalInferenceSetup.identify_failure_points
6487:         Step 5: Dynamics pattern - CausalInferenceSetup._get_dynamics_pattern
6488:         Step 6: Text chunking - SemanticProcessor.chunk_text
6489:         Step 7: PDM structure detection - SemanticProcessor._detect_pdm_structure
6490:         Step 8: Table detection - SemanticProcessor._detect_table
6491:         Step 9: Traceability record - AdaptivePriorCalculator.generate_traceability_record

```

```
6492: """
6493:
6494:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
6495:         raw_evidence = {}
6496:
6497:         # Step 1: Calculate language specificity
6498:         language_specificity = self._execute_method(
6499:             "CausalExtractor", "_calculate_language_specificity", context
6500:         )
6501:
6502:         # Step 2: Assess temporal coherence
6503:         temporal_coherence = self._execute_method(
6504:             "CausalExtractor", "_assess_temporal_coherence", context
6505:         )
6506:
6507:         # Step 3: Diagnose critical links
6508:         critical_links = self._execute_method(
6509:             "TextMiningEngine", "diagnose_critical_links", context
6510:         )
6511:
6512:         # Step 4: Identify failure points
6513:         failure_points = self._execute_method(
6514:             "CausalInferenceSetup", "identify_failure_points", context
6515:         )
6516:
6517:         # Step 5: Get dynamics pattern
6518:         dynamics_pattern = self._execute_method(
6519:             "CausalInferenceSetup", "_get_dynamics_pattern", context
6520:         )
6521:
6522:         # Step 6: Process text structure
6523:         text_chunks = self._execute_method(
6524:             "SemanticProcessor", "chunk_text", context
6525:         )
6526:         pdm_structure = self._execute_method(
6527:             "SemanticProcessor", "_detect_pdm_structure", context,
6528:             chunks=text_chunks
6529:         )
6530:         table_detection = self._execute_method(
6531:             "SemanticProcessor", "_detect_table", context,
6532:             chunks=text_chunks
6533:         )
6534:
6535:         # Step 7: Generate traceability record
6536:         traceability = self._execute_method(
6537:             "AdaptivePriorCalculator", "generate_traceability_record", context,
6538:             specificity=language_specificity
6539:         )
6540:
6541:         raw_evidence = {
6542:             "context_adaptation": (language_specificity or {}).get("adaptation_level", 0),
6543:             "differential_impacts_recognized": (critical_links or {}).get("differential_groups", []),
6544:             "specific_groups_mentioned": (critical_links or {}).get("target_groups", []),
6545:             "territorial_constraints": (failure_points or {}).get("territorial", []),
6546:             "local_context_integration": (pdm_structure or {}).get("local_sections", []),
6547:             "language_specificity": language_specificity,
```

```
6548:         "temporal_coherence": temporal_coherence,
6549:         "critical_links": critical_links,
6550:         "failure_points": failure_points,
6551:         "dynamics_pattern": dynamics_pattern,
6552:         "structure_analysis": pdm_structure,
6553:         "table_detection": table_detection,
6554:         "text_chunks": text_chunks,
6555:         "traceability": traceability
6556:     }
6557:
6558:     return {
6559:         "executor_id": self.executor_id,
6560:         "raw_evidence": raw_evidence,
6561:         "metadata": {
6562:             "methods_executed": [log["method"] for log in self.execution_log],
6563:             "groups_identified": len((critical_links or {}).get("target_groups", [])) if critical_links else 0,
6564:             "territorial_constraints": len((failure_points or {}).get("territorial", [])) if failure_points else 0,
6565:             "total_text_chunks": len(text_chunks) if text_chunks else 0,
6566:             "has_text_chunks": bool(text_chunks),
6567:             "total_table_detections": len(table_detection) if isinstance(table_detection, list) else (1 if table_detection else 0),
6568:             "has_table_detection": bool(table_detection),
6569:             "has_pdm_structure": bool(pdm_structure),
6570:             "has_dynamics_pattern": bool(dynamics_pattern),
6571:             "has_traceability": bool(traceability)
6572:         },
6573:         "execution_metrics": {
6574:             "methods_count": len(self.execution_log),
6575:             "all_succeeded": all(log["success"] for log in self.execution_log)
6576:         }
6577:     }
6578:
6579:
6580: # =====
6581: # EXECUTOR REGISTRY
6582: # =====
6583:
6584: EXECUTOR_REGISTRY = {
6585:     "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
6586:     "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
6587:     "D1-Q3": D1_Q3_BudgetAllocationTracer,
6588:     "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
6589:     "D1-Q5": D1_Q5_ScopeJustificationValidator,
6590:
6591:     "D2-Q1": D2_Q1_StructuredPlanningValidator,
6592:     "D2-Q2": D2_Q2_InterventionLogicInferencer,
6593:     "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
6594:     "D2-Q4": D2_Q4_RiskManagementAnalyzer,
6595:     "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,
6596:
6597:     "D3-Q1": D3_Q1_IndicatorQualityValidator,
6598:     "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
6599:     "D3-Q3": D3_Q3_TraceabilityValidator,
6600:     "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
6601:     "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,
6602:
6603:     "D4-Q1": D4_Q1_OutcomeMetricsValidator,
```

```
6604:     "D4-Q2": D4_Q2_CausalChainValidator,
6605:     "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
6606:     "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
6607:     "D4-Q5": D4_Q5_VerticalAlignmentValidator,
6608:
6609:     "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
6610:     "D5-Q2": D5_Q2_CompositeMeasurementValidator,
6611:     "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
6612:     "D5-Q4": D5_Q4_SystemicRiskEvaluator,
6613:     "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,
6614:
6615:     "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
6616:     "D6-Q2": D6_Q2_LogicalProportionalityValidator,
6617:     "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
6618:     "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
6619:     "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
6620: }
6621:
6622:
6623: # =====
6624: # PHASE 2 ORCHESTRATION
6625: # =====
6626:
6627:
6628: def _build_method_executor() -> MethodExecutor:
6629:     """Construct a canonical MethodExecutor via the factory wiring."""
6630:     bundle = build_processor()
6631:     method_executor = getattr(bundle, "method_executor", None)
6632:     if not isinstance(method_executor, MethodExecutor):
6633:         raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor instance.")
6634:     return method_executor
6635:
6636:
6637: def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
6638:     """Build canonical metadata block using canonical_notation."""
6639:     metadata: Dict[str, Any] = {}
6640:     try:
6641:         dim_key = executor_id.split("-")[0]
6642:         dim_info = get_dimension_info(dim_key)
6643:         metadata["dimension_code"] = dim_info.code
6644:         metadata["dimension_label"] = dim_info.label
6645:     except (KeyError, ValueError, IndexError, AttributeError) as e:
6646:         logger.warning(f"Failed to load canonical metadata for {executor_id}: {e}")
6647:         # Continue with empty metadata rather than failing
6648:     # Let critical system exceptions (KeyboardInterrupt, SystemExit) propagate
6649:
6650:     if executor_id in CANONICAL_QUESTION_LABELS:
6651:         metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
6652:     return metadata
6653:
6654:
6655: def run_phase2_executors(context_package: Dict[str, Any],
6656:                           policy_areas: List[str]) -> Dict[str, Any]:
6657:     """
6658:     Phase 2 Entry Point: Runs all 30 executors for each policy area.
6659:
```

```
6660:     Args:
6661:         context_package: Canonical package with document data from Phase 1
6662:         policy_areas: List of policy area identifiers to analyze
6663:
6664:     Returns:
6665:         Dict mapping policy_area -> executor_id -> raw_evidence
6666:         """
6667:     results = {}
6668:     method_executor = _build_method_executor()
6669:
6670:     for policy_area in policy_areas:
6671:         print(f"\n{'='*80}")
6672:         print(f"Processing Policy Area: {policy_area}")
6673:         print(f"{'='*80}\n")
6674:
6675:         # Prepare context for this policy area
6676:         area_context = {
6677:             **context_package,
6678:             "policy_area": policy_area
6679:         }
6680:
6681:         # Execute all 30 executors
6682:         area_results = {}
6683:         for executor_id, executor_class in EXECUTOR_REGISTRY.items():
6684:             print(f"Running {executor_id}: {executor_class.__name__}...")
6685:
6686:             try:
6687:                 # Instantiate executor with config
6688:                 config = load_executor_config(executor_id)
6689:                 executor = executor_class(executor_id, config, method_executor=method_executor)
6690:
6691:                 # Execute and collect results
6692:                 result = executor.execute(area_context)
6693:                 # Append canonical metadata consistently
6694:                 result_metadata = result.get("metadata", {})
6695:                 result_metadata.update(_canonical_metadata(executor_id))
6696:                 result["metadata"] = result_metadata
6697:                 area_results[executor_id] = result
6698:
6699:                 print(f" \u2708\ufe0f Success: {len(result['metadata']['methods_executed'])} methods executed")
6700:
6701:             except ExecutorFailure as e:
6702:                 print(f" \u2708\ufe0f FAILED: {str(e)}")
6703:                 raise # Re-raise to stop execution as per requirement
6704:
6705:             results[policy_area] = area_results
6706:
6707:     return results
6708:
6709:
6710: def load_executor_config(executor_id: str) -> Dict[str, Any]:
6711:     """
6712:     Load executor configuration from JSON contract.
6713:
6714:     Args:
6715:         executor_id: Executor identifier (e.g., "D1-Q1")
```

```
6716:  
6717:     Returns:  
6718:         Configuration dictionary from JSON contract  
6719:     """  
6720:     import json  
6721:     from pathlib import Path  
6722:  
6723:     config_path = Path(f"config/executor_contracts/{executor_id}.json")  
6724:  
6725:     if not config_path.exists():  
6726:         raise FileNotFoundError(f"Executor config not found: {config_path}")  
6727:  
6728:     with open(config_path, 'r', encoding='utf-8') as f:  
6729:         return json.load(f)  
6730:  
6731:  
6732: # =====  
6733: # EXAMPLE USAGE  
6734: # =====  
6735:  
6736: if __name__ == "__main__":  
6737:     # Example context package from Phase 1  
6738:     context_package = {  
6739:         "document_path": "data/pdm_municipality_xyz.pdf",  
6740:         "document_text": "...", # Full document text  
6741:         "tables": [], # Extracted tables from Phase 1  
6742:         "embeddings": {}, # Precomputed embeddings  
6743:         "entities": [], # Pre-extracted entities  
6744:         "metadata": {  
6745:             "municipality": "Municipality XYZ",  
6746:             "year": 2024,  
6747:             "pages": 150  
6748:         }  
6749:     }  
6750:  
6751:     # Policy areas to analyze  
6752:     policy_areas = [  
6753:         "PA01", # Education  
6754:         "PA02", # Health  
6755:         "PA03", # Infrastructure  
6756:         # ... up to 10+ policy areas  
6757:     ]  
6758:  
6759:     # Run Phase 2  
6760:     try:  
6761:         results = run_phase2_executors(context_package, policy_areas)  
6762:         print("\n" + "="*80)  
6763:         print("PHASE 2 COMPLETED SUCCESSFULLY")  
6764:         print("="*80)  
6765:         print(f"Processed {len(policy_areas)} policy areas")  
6766:         print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")  
6767:         print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")  
6768:  
6769:     except ExecutorFailure as e:  
6770:         print("\n" + "="*80)  
6771:         print("PHASE 2 FAILED")
```

```
6772:     print("*"*80)
6773:     print(f"Error: {str(e)}")
6774:     print("Execution halted as per requirement: any method failure = executor failure")
6775:
6776:
6777:
6778: =====
6779: FILE: src/farfan_pipeline/core/orchestrator/executors_contract.py
6780: =====
6781:
6782: from __future__ import annotations
6783:
6784: from farfan_pipeline.core.orchestrator.base_executor_with_contract import BaseExecutorWithContract
6785:
6786:
6787: class D1Q1_Executor_Contract(BaseExecutorWithContract):
6788:     @classmethod
6789:     def get_base_slot(cls) -> str:
6790:         return "D1-Q1"
6791:
6792:
6793: class D1Q2_Executor_Contract(BaseExecutorWithContract):
6794:     @classmethod
6795:     def get_base_slot(cls) -> str:
6796:         return "D1-Q2"
6797:
6798:
6799: class D1Q3_Executor_Contract(BaseExecutorWithContract):
6800:     @classmethod
6801:     def get_base_slot(cls) -> str:
6802:         return "D1-Q3"
6803:
6804:
6805: class D1Q4_Executor_Contract(BaseExecutorWithContract):
6806:     @classmethod
6807:     def get_base_slot(cls) -> str:
6808:         return "D1-Q4"
6809:
6810:
6811: class D1Q5_Executor_Contract(BaseExecutorWithContract):
6812:     @classmethod
6813:     def get_base_slot(cls) -> str:
6814:         return "D1-Q5"
6815:
6816:
6817: class D2Q1_Executor_Contract(BaseExecutorWithContract):
6818:     @classmethod
6819:     def get_base_slot(cls) -> str:
6820:         return "D2-Q1"
6821:
6822:
6823: class D2Q2_Executor_Contract(BaseExecutorWithContract):
6824:     @classmethod
6825:     def get_base_slot(cls) -> str:
6826:         return "D2-Q2"
6827:
```

```
6828:
6829: class D2Q3_Executor_Contract(BaseExecutorWithContract):
6830:     @classmethod
6831:     def get_base_slot(cls) -> str:
6832:         return "D2-Q3"
6833:
6834:
6835: class D2Q4_Executor_Contract(BaseExecutorWithContract):
6836:     @classmethod
6837:     def get_base_slot(cls) -> str:
6838:         return "D2-Q4"
6839:
6840:
6841: class D2Q5_Executor_Contract(BaseExecutorWithContract):
6842:     @classmethod
6843:     def get_base_slot(cls) -> str:
6844:         return "D2-Q5"
6845:
6846:
6847: class D3Q1_Executor_Contract(BaseExecutorWithContract):
6848:     @classmethod
6849:     def get_base_slot(cls) -> str:
6850:         return "D3-Q1"
6851:
6852:
6853: class D3Q2_Executor_Contract(BaseExecutorWithContract):
6854:     @classmethod
6855:     def get_base_slot(cls) -> str:
6856:         return "D3-Q2"
6857:
6858:
6859: class D3Q3_Executor_Contract(BaseExecutorWithContract):
6860:     @classmethod
6861:     def get_base_slot(cls) -> str:
6862:         return "D3-Q3"
6863:
6864:
6865: class D3Q4_Executor_Contract(BaseExecutorWithContract):
6866:     @classmethod
6867:     def get_base_slot(cls) -> str:
6868:         return "D3-Q4"
6869:
6870:
6871: class D3Q5_Executor_Contract(BaseExecutorWithContract):
6872:     @classmethod
6873:     def get_base_slot(cls) -> str:
6874:         return "D3-Q5"
6875:
6876:
6877: class D4Q1_Executor_Contract(BaseExecutorWithContract):
6878:     @classmethod
6879:     def get_base_slot(cls) -> str:
6880:         return "D4-Q1"
6881:
6882:
6883: class D4Q2_Executor_Contract(BaseExecutorWithContract):
```

```
6884:     @classmethod
6885:     def get_base_slot(cls) -> str:
6886:         return "D4-Q2"
6887:
6888:
6889: class D4Q3_Executor_Contract(BaseExecutorWithContract):
6890:     @classmethod
6891:     def get_base_slot(cls) -> str:
6892:         return "D4-Q3"
6893:
6894:
6895: class D4Q4_Executor_Contract(BaseExecutorWithContract):
6896:     @classmethod
6897:     def get_base_slot(cls) -> str:
6898:         return "D4-Q4"
6899:
6900:
6901: class D4Q5_Executor_Contract(BaseExecutorWithContract):
6902:     @classmethod
6903:     def get_base_slot(cls) -> str:
6904:         return "D4-Q5"
6905:
6906:
6907: class D5Q1_Executor_Contract(BaseExecutorWithContract):
6908:     @classmethod
6909:     def get_base_slot(cls) -> str:
6910:         return "D5-Q1"
6911:
6912:
6913: class D5Q2_Executor_Contract(BaseExecutorWithContract):
6914:     @classmethod
6915:     def get_base_slot(cls) -> str:
6916:         return "D5-Q2"
6917:
6918:
6919: class D5Q3_Executor_Contract(BaseExecutorWithContract):
6920:     @classmethod
6921:     def get_base_slot(cls) -> str:
6922:         return "D5-Q3"
6923:
6924:
6925: class D5Q4_Executor_Contract(BaseExecutorWithContract):
6926:     @classmethod
6927:     def get_base_slot(cls) -> str:
6928:         return "D5-Q4"
6929:
6930:
6931: class D5Q5_Executor_Contract(BaseExecutorWithContract):
6932:     @classmethod
6933:     def get_base_slot(cls) -> str:
6934:         return "D5-Q5"
6935:
6936:
6937: class D6Q1_Executor_Contract(BaseExecutorWithContract):
6938:     @classmethod
6939:     def get_base_slot(cls) -> str:
```

```
6940:         return "D6-Q1"
6941:
6942:
6943: class D6Q2_Executor_Contract(BaseExecutorWithContract):
6944:     @classmethod
6945:     def get_base_slot(cls) -> str:
6946:         return "D6-Q2"
6947:
6948:
6949: class D6Q3_Executor_Contract(BaseExecutorWithContract):
6950:     @classmethod
6951:     def get_base_slot(cls) -> str:
6952:         return "D6-Q3"
6953:
6954:
6955: class D6Q4_Executor_Contract(BaseExecutorWithContract):
6956:     @classmethod
6957:     def get_base_slot(cls) -> str:
6958:         return "D6-Q4"
6959:
6960:
6961: class D6Q5_Executor_Contract(BaseExecutorWithContract):
6962:     @classmethod
6963:     def get_base_slot(cls) -> str:
6964:         return "D6-Q5"
6965:
6966:
6967: # Aliases expected by core orchestrator
6968: D1Q1_Executor = D1Q1_Executor_Contract
6969: D1Q2_Executor = D1Q2_Executor_Contract
6970: D1Q3_Executor = D1Q3_Executor_Contract
6971: D1Q4_Executor = D1Q4_Executor_Contract
6972: D1Q5_Executor = D1Q5_Executor_Contract
6973: D2Q1_Executor = D2Q1_Executor_Contract
6974: D2Q2_Executor = D2Q2_Executor_Contract
6975: D2Q3_Executor = D2Q3_Executor_Contract
6976: D2Q4_Executor = D2Q4_Executor_Contract
6977: D2Q5_Executor = D2Q5_Executor_Contract
6978: D3Q1_Executor = D3Q1_Executor_Contract
6979: D3Q2_Executor = D3Q2_Executor_Contract
6980: D3Q3_Executor = D3Q3_Executor_Contract
6981: D3Q4_Executor = D3Q4_Executor_Contract
6982: D3Q5_Executor = D3Q5_Executor_Contract
6983: D4Q1_Executor = D4Q1_Executor_Contract
6984: D4Q2_Executor = D4Q2_Executor_Contract
6985: D4Q3_Executor = D4Q3_Executor_Contract
6986: D4Q4_Executor = D4Q4_Executor_Contract
6987: D4Q5_Executor = D4Q5_Executor_Contract
6988: D5Q1_Executor = D5Q1_Executor_Contract
6989: D5Q2_Executor = D5Q2_Executor_Contract
6990: D5Q3_Executor = D5Q3_Executor_Contract
6991: D5Q4_Executor = D5Q4_Executor_Contract
6992: D5Q5_Executor = D5Q5_Executor_Contract
6993: D6Q1_Executor = D6Q1_Executor_Contract
6994: D6Q2_Executor = D6Q2_Executor_Contract
6995: D6Q3_Executor = D6Q3_Executor_Contract
```

```
6996: D6Q4_Executor = D6Q4_Executor_Contract
6997: D6Q5_Executor = D6Q5_Executor_Contract
6998:
6999:
7000:
7001: =====
7002: FILE: src/farfan_pipeline/core/orchestrator/executors_snapshot/executors.py
7003: =====
7004:
7005: """
7006: executors.py - Phase 2: Executor Orchestration for Policy Document Analysis
7007:
7008: This module defines 30 executors (one per D{n}-Q{m} question) that orchestrate
7009: methods from the core module to extract raw evidence from Colombian municipal
7010: development plans (PDET/PDM documents).
7011:
7012: Architecture:
7013: - Each executor is independent and receives a canonical context package
7014: - Methods execute in configured order; any failure causes executor failure
7015: - Outputs are Python dicts/lists matching JSON contract specifications
7016: - Executors are injected via MethodExecutor factory pattern
7017:
7018: Usage:
7019:     from factory import run_executor
7020:     result = run_executor("D1-Q1", context_package)
7021: """
7022:
7023: from typing import Dict, List, Any, Optional
7024: from abc import ABC, abstractmethod
7025:
7026: from farfan_pipeline.core.canonical_notation import CanonicalDimension, get_dimension_info
7027: from farfan_pipeline.core.orchestrator.core import MethodExecutor
7028: from farfan_pipeline.core.orchestrator.factory import build_processor
7029:
7030: # Canonical question labels (only defined when verified in repo)
7031: CANONICAL_QUESTION_LABELS = {
7032:     "D3-Q2": "DIM03_Q02_PRODUCT_TARGET_PROPORIONALITY",
7033:     "D3-Q3": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
7034:     "D3-Q4": "DIM03_Q04_TECHNICAL_FEASIBILITY",
7035:     "D3-Q5": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
7036:     "D4-Q1": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
7037:     "D5-Q2": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
7038: }
7039:
7040: # Epistemic taxonomy per method (focused on executors expanded in this iteration)
7041: EPISTEMIC_TAGS = {
7042:     ("FinancialAuditor", "_calculate_sufficiency"): ["statistical", "normative"],
7043:     ("FinancialAuditor", "_match_program_to_node"): ["structural"],
7044:     ("FinancialAuditor", "_match_goal_to_budget"): ["structural", "normative"],
7045:     ("PDET MunicipalPlanAnalyzer", "_assess_financial_sustainability"): ["financial", "normative"],
7046:     ("PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility"): ["financial", "statistical"],
7047:     ("PDET MunicipalPlanAnalyzer", "_score_indicators"): ["normative", "semantic"],
7048:     ("PDET MunicipalPlanAnalyzer", "_interpret_risk"): ["normative", "statistical"],
7049:     ("PDET MunicipalPlanAnalyzer", "_extract_from_responsibility_tables"): ["structural"],
7050:     ("PDET MunicipalPlanAnalyzer", "_consolidate_entities"): ["structural"],
7051:     ("PDET MunicipalPlanAnalyzer", "_extract_entities_syntax"): ["semantic"],
```

```
7052: ("PDET MunicipalPlanAnalyzer", "_extract_entities_ner"): ["semantic"],
7053: ("PDET MunicipalPlanAnalyzer", "identify_responsible_entities"): ["semantic", "structural"],
7054: ("PDET MunicipalPlanAnalyzer", "_score_responsibility_clarity"): ["normative"],
7055: ("PDET MunicipalPlanAnalyzer", "_refine_edge_probabilities"): ["statistical", "causal"],
7056: ("PDET MunicipalPlanAnalyzer", "construct_causal_dag"): ["structural", "causal"],
7057: ("PDET MunicipalPlanAnalyzer", "estimate_causal_effects"): ["causal", "statistical"],
7058: ("PDET MunicipalPlanAnalyzer", "generate_counterfactuals"): ["causal"],
7059: ("PDET MunicipalPlanAnalyzer", "_identify_confounders"): ["causal", "consistency"],
7060: ("PDET MunicipalPlanAnalyzer", "_effect_to_dict"): ["descriptive"],
7061: ("PDET MunicipalPlanAnalyzer", "_scenario_to_dict"): ["descriptive"],
7062: ("PDET MunicipalPlanAnalyzer", "_get_spanish_stopwords"): ["semantic"],
7063: ("AdaptivePriorCalculator", "calculate_likelihood_adaptativo"): ["statistical", "bayesian"],
7064: ("AdaptivePriorCalculator", "_adjust_domain_weights"): ["statistical"],
7065: ("BayesianMechanismInference", "_test_sufficiency"): ["statistical", "bayesian"],
7066: ("BayesianMechanismInference", "_test_necessity"): ["statistical", "bayesian"],
7067: ("BayesianMechanismInference", "_log_refactored_components"): ["implementation"],
7068: ("BayesianMechanismInference", "_infer_activity_sequence"): ["causal"],
7069: ("BayesianMechanismInference", "infer_mechanisms"): ["causal", "bayesian"],
7070: ("AdvancedDAGValidator", "calculate_acyclicity_pvalue"): ["statistical", "consistency"],
7071: ("AdvancedDAGValidator", "_is_acyclic"): ["structural", "consistency"],
7072: ("AdvancedDAGValidator", "_calculate_bayesian_posterior"): ["statistical", "bayesian"],
7073: ("AdvancedDAGValidator", "_calculate_confidence_interval"): ["statistical"],
7074: ("AdvancedDAGValidator", "_calculate_statistical_power"): ["statistical"],
7075: ("AdvancedDAGValidator", "_generate_subgraph"): ["structural"],
7076: ("AdvancedDAGValidator", "_get_node_validator"): ["implementation"],
7077: ("AdvancedDAGValidator", "_create_empty_result"): ["descriptive"],
7078: ("AdvancedDAGValidator", "_initialize_rng"): ["implementation"],
7079: ("AdvancedDAGValidator", "get_graph_stats"): ["structural"],
7080: ("AdvancedDAGValidator", "_calculate_node_importance"): ["structural"],
7081: ("AdvancedDAGValidator", "export_nodes"): ["structural", "descriptive"],
7082: ("AdvancedDAGValidator", "add_node"): ["structural"],
7083: ("AdvancedDAGValidator", "add_edge"): ["structural"],
7084: ("IndustrialGradeValidator", "execute_suite"): ["implementation", "normative"],
7085: ("IndustrialGradeValidator", "validate_connection_matrix"): ["consistency"],
7086: ("IndustrialGradeValidator", "run_performance_benchmarks"): ["implementation"],
7087: ("IndustrialGradeValidator", "_benchmark_operation"): ["implementation"],
7088: ("IndustrialGradeValidator", "validate_causal_categories"): ["consistency"],
7089: ("IndustrialGradeValidator", "_log_metric"): ["implementation"],
7090: ("PerformanceAnalyzer", "analyze_performance"): ["implementation", "normative"],
7091: ("PerformanceAnalyzer", "_calculate_loss_functions"): ["statistical"],
7092: ("HierarchicalGenerativeModel", "_calculate_ess"): ["statistical"],
7093: ("HierarchicalGenerativeModel", "_calculate_likelihood"): ["statistical"],
7094: ("HierarchicalGenerativeModel", "_calculate_r_hat"): ["statistical"],
7095: ("ReportingEngine", "generate_accountability_matrix"): ["normative", "structural"],
7096: ("ReportingEngine", "_calculate_quality_score"): ["normative", "statistical"],
7097: ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "descriptive"],
7098: ("PolicyAnalysisEmbedder", "compare_policy_interventions"): ["normative"],
7099: ("PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency"): ["consistency", "statistical"],
7100: ("PolicyAnalysisEmbedder", "process_document"): ["semantic", "structural"],
7101: ("PolicyAnalysisEmbedder", "semantic_search"): ["semantic"],
7102: ("PolicyAnalysisEmbedder", "_apply_mmr"): ["semantic"],
7103: ("PolicyAnalysisEmbedder", "_generate_query_from_pdq"): ["semantic"],
7104: ("PolicyAnalysisEmbedder", "_filter_by_pdq"): ["semantic"],
7105: ("PolicyAnalysisEmbedder", "_extract_numerical_values"): ["statistical"],
7106: ("PolicyAnalysisEmbedder", "_compute_overall_confidence"): ["statistical", "normative"],
7107: ("PolicyAnalysisEmbedder", "_embed_texts"): ["semantic"],
```

```
7108: ("SemanticAnalyzer", "_classify_policy_domain"): ["semantic"],
7109: ("SemanticAnalyzer", "_empty_semantic_cube"): ["descriptive"],
7110: ("SemanticAnalyzer", "_classify_cross_cutting_themes"): ["semantic"],
7111: ("SemanticAnalyzer", "_classify_value_chain_link"): ["semantic"],
7112: ("SemanticAnalyzer", "_vectorize_segments"): ["semantic"],
7113: ("SemanticAnalyzer", "_calculate_semantic_complexity"): ["semantic"],
7114: ("SemanticAnalyzer", "_process_segment"): ["semantic"],
7115: ("PDET MunicipalPlanAnalyzer", "_entity_to_dict"): ["descriptive"],
7116: ("PDET MunicipalPlanAnalyzer", "_quality_to_dict"): ["descriptive", "normative"],
7117: ("PDET MunicipalPlanAnalyzer", "_deduplicate_tables"): ["structural", "implementation"],
7118: ("PDET MunicipalPlanAnalyzer", "_indicator_to_dict"): ["descriptive"],
7119: ("PDET MunicipalPlanAnalyzer", "_generate_recommendations"): ["normative"],
7120: ("PDET MunicipalPlanAnalyzer", "_simulate_intervention"): ["causal", "statistical"],
7121: ("PDET MunicipalPlanAnalyzer", "_identify_causal_nodes"): ["structural", "causal"],
7122: ("PDET MunicipalPlanAnalyzer", "_match_text_to_node"): ["semantic", "structural"],
7123: ("TeoriaCambio", "_validar_orden_causal"): ["causal", "consistency"],
7124: ("TeoriaCambio", "_generar_sugerencias_internas"): ["normative"],
7125: ("TeoriaCambio", "_extraer_categorias"): ["semantic"],
7126: ("BayesianMechanismInference", "_extract_observations"): ["semantic", "causal"],
7127: ("BayesianMechanismInference", "_generate_necessity_remediation"): ["normative", "causal"],
7128: ("BayesianMechanismInference", "_quantify_uncertainty"): ["statistical", "bayesian"],
7129: ("CausalExtractor", "_build_type_hierarchy"): ["structural"],
7130: ("CausalExtractor", "_check_structuralViolation"): ["structural", "consistency"],
7131: ("CausalExtractor", "_calculate_type_transition_prior"): ["statistical", "bayesian"],
7132: ("CausalExtractor", "_calculate_textual_proximity"): ["semantic"],
7133: ("CausalExtractor", "_calculate_language_specificity"): ["semantic"],
7134: ("CausalExtractor", "_calculate_composite_likelihood"): ["statistical", "semantic"],
7135: ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
7136: ("CausalExtractor", "_calculate_semantic_distance"): ["semantic"],
7137: ("CausalExtractor", "_extract_goals"): ["semantic"],
7138: ("CausalExtractor", "_parse_goal_context"): ["semantic"],
7139: ("CausalExtractor", "_classify_goal_type"): ["semantic"],
7140: ("TemporalLogicVerifier", "_parse_temporal_marker"): ["temporal", "consistency"],
7141: ("TemporalLogicVerifier", "_classify_temporal_type"): ["temporal", "consistency"],
7142: ("TemporalLogicVerifier", "_extract_resources"): ["structural"],
7143: ("TemporalLogicVerifier", "_should_precede"): ["temporal", "consistency"],
7144: ("AdaptivePriorCalculator", "generate_traceability_record"): ["structural", "semantic"],
7145: ("PolicyAnalysisEmbedder", "generate_pdq_report"): ["semantic", "normative"],
7146: ("ReportingEngine", "generate_confidence_report"): ["normative", "descriptive"],
7147: ("PolicyTextProcessor", "segment_into_sentences"): ["semantic", "structural"],
7148: ("PolicyTextProcessor", "normalize_unicode"): ["implementation"],
7149: ("PolicyTextProcessor", "compile_pattern"): ["implementation"],
7150: ("PolicyTextProcessor", "extract_contextual_window"): ["semantic"],
7151: ("BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize"): ["causal", "normative"],
7152: ("BayesianCounterfactualAuditor", "refutation_and_sanity_checks"): ["causal", "consistency"],
7153: ("BayesianCounterfactualAuditor", "_evaluate_factual"): ["causal", "statistical"],
7154: ("BayesianCounterfactualAuditor", "_evaluate_counterfactual"): ["causal", "statistical"],
7155: ("CausalExtractor", "_assess_financial_consistency"): ["financial", "consistency"],
7156: ("IndustrialPolicyProcessor", "_load_questionnaire"): ["descriptive", "implementation"],
7157: ("IndustrialPolicyProcessor", "_compile_pattern_registry"): ["structural", "semantic"],
7158: ("IndustrialPolicyProcessor", "_build_point_patterns"): ["semantic"],
7159: ("IndustrialPolicyProcessor", "_empty_result"): ["implementation"],
7160: ("IndustrialPolicyProcessor", "_compute_evidence_confidence"): ["statistical"],
7161: ("IndustrialPolicyProcessor", "_compute_avg_confidence"): ["statistical"],
7162: ("IndustrialPolicyProcessor", "_construct_evidence_bundle"): ["structural"],
7163: ("PDET MunicipalPlanAnalyzer", "generate_executive_report"): ["normative"],
```

```
7164:     ("IndustrialPolicyProcessor", "export_results"): ["implementation"],
7165: }
7166:
7167:
7168: class BaseExecutor(ABC):
7169: """
7170:     Base class for all executors with standardized execution template.
7171:     All executors must implement execute() and return structured evidence.
7172: """
7173:
7174:     def __init__(self, executor_id: str, config: Dict[str, Any], method_executor: MethodExecutor):
7175:         self.executor_id = executor_id
7176:         self.config = config
7177:         if not isinstance(method_executor, MethodExecutor):
7178:             raise RuntimeError("A valid MethodExecutor instance is required for executor injection.")
7179:         self.method_executor = method_executor
7180:         self.execution_log = []
7181:         self.dimension_info = None
7182:         try:
7183:             dim_key = executor_id.split("-")[0].replace("D", "D")
7184:             self.dimension_info = get_dimension_info(dim_key)
7185:         except Exception:
7186:             self.dimension_info = None
7187:
7188:     @abstractmethod
7189:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7190: """
7191:     Execute configured methods and return raw evidence.
7192:
7193:     Args:
7194:         context: Canonical package with document, tables, metadata
7195:
7196:     Returns:
7197:         Dict with raw_evidence, metadata, execution_metrics
7198:
7199:     Raises:
7200:         ExecutorFailure: If any method fails
7201: """
7202:     pass
7203:
7204:     def _log_method_execution(self, class_name: str, method_name: str,
7205:                             success: bool, result: Any = None, error: str = None):
7206: """
7207:     Track method execution for debugging and traceability."""
7208:         self.execution_log.append({
7209:             "class": class_name,
7210:             "method": method_name,
7211:             "success": success,
7212:             "result_type": type(result).__name__ if result else None,
7213:             "error": error
7214:         })
7215:     def _execute_method(self, class_name: str, method_name: str,
7216:                         context: Dict[str, Any], **kwargs) -> Any:
7217: """
7218:     Execute a single method with error handling.
7219:
```

```
7220:     Raises:
7221:         ExecutorFailure: If method execution fails
7222:     """
7223:     try:
7224:         # Method injection happens via factory - placeholder for actual execution
7225:         method = self._get_method(class_name, method_name)
7226:         result = method(context, **kwargs)
7227:         self._log_method_execution(class_name, method_name, True, result)
7228:         return result
7229:     except Exception as e:
7230:         self._log_method_execution(class_name, method_name, False, error=str(e))
7231:         raise ExecutorFailure(
7232:             f"Executor {self.executor_id} failed: {class_name}.{method_name} - {str(e)}"
7233:         )
7234:
7235:     def _get_method(self, class_name: str, method_name: str):
7236:         """Retrieve method using MethodExecutor to enforce routed execution."""
7237:         if not isinstance(self.method_executor, MethodExecutor):
7238:             raise RuntimeError(f"Invalid method executor provided: {type(self.method_executor).__name__}")
7239:
7240:     def _wrapped(context: Dict[str, Any], **kwargs: Any) -> Any:
7241:         payload: Dict[str, Any] = {}
7242:         if context:
7243:             payload.update(context)
7244:         if kwargs:
7245:             payload.update(kwargs)
7246:         return self.method_executor.execute(
7247:             class_name=class_name,
7248:             method_name=method_name,
7249:             **payload,
7250:         )
7251:
7252:     return _wrapped
7253:
7254:
7255: class ExecutorFailure(Exception):
7256:     """Raised when any method in an executor fails."""
7257:     pass
7258:
7259:
7260: # =====
7261: # DIMENSION 1: DIAGNOSTICS & INPUTS
7262: # =====
7263:
7264: class D1_Q1_QuantitativeBaselineExtractor(BaseExecutor):
7265:     """
7266:         Extracts numeric data, reference years, and official sources as baseline.
7267:
7268:         Methods (from D1-Q1):
7269:         - TextMiningEngine.diagnose_critical_links
7270:         - TextMiningEngine._analyze_link_text
7271:         - IndustrialPolicyProcessor.process
7272:         - IndustrialPolicyProcessor._match_patterns_in_sentences
7273:         - IndustrialPolicyProcessor._extract_point_evidence
7274:         - CausalExtractor._extract_goals
7275:         - CausalExtractor._parse_goal_context

```

```
7276:     - FinancialAuditor._parse_amount
7277:     - PDET MunicipalPlanAnalyzer._extract_financial_amounts
7278:     - PDET MunicipalPlanAnalyzer._extract_from_budget_table
7279:     - PolicyContradictionDetector._extract_quantitative_claims
7280:     - PolicyContradictionDetector._parse_number
7281:     - PolicyContradictionDetector._statistical_significance_test
7282:     - BayesianNumericalAnalyzer.evaluate_policy_metric
7283:     - BayesianNumericalAnalyzer.compare_policies
7284:     - SemanticProcessor.chunk_text
7285:     - SemanticProcessor.embed_single
7286: """
7287:
7288: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7289:     raw_evidence = {}
7290:
7291:     # Step 1: Identify critical data-bearing sections
7292:     critical_links = self._execute_method(
7293:         "TextMiningEngine", "diagnose_critical_links", context
7294:     )
7295:     link_analysis = self._execute_method(
7296:         "TextMiningEngine", "_analyze_link_text", context,
7297:         links=critical_links
7298:     )
7299:
7300:     # Step 2: Extract structured quantitative claims
7301:     processed_sections = self._execute_method(
7302:         "IndustrialPolicyProcessor", "process", context
7303:     )
7304:     pattern_matches = self._execute_method(
7305:         "IndustrialPolicyProcessor", "_match_patterns_in_sentences", context,
7306:         sections=processed_sections
7307:     )
7308:     point_evidence = self._execute_method(
7309:         "IndustrialPolicyProcessor", "_extract_point_evidence", context,
7310:         matches=pattern_matches
7311:     )
7312:
7313:     # Step 3: Parse numerical amounts and baseline data
7314:     parsed_amounts = self._execute_method(
7315:         "FinancialAuditor", "_parse_amount", context,
7316:         evidence=point_evidence
7317:     )
7318:     financial_amounts = self._execute_method(
7319:         "PDET MunicipalPlanAnalyzer", "_extract_financial_amounts", context
7320:     )
7321:     budget_table_data = self._execute_method(
7322:         "PDET MunicipalPlanAnalyzer", "_extract_from_budget_table", context
7323:     )
7324:
7325:     # Step 4: Extract temporal context (reference years)
7326:     goals = self._execute_method(
7327:         "CausalExtractor", "_extract_goals", context
7328:     )
7329:     goal_contexts = self._execute_method(
7330:         "CausalExtractor", "_parse_goal_context", context,
7331:         goals=goals
```

```
7332:     )
7333:
7334:     # Step 5: Validate quantitative claims
7335:     quant_claims = self._execute_method(
7336:         "PolicyContradictionDetector", "_extract_quantitative_claims", context
7337:     )
7338:     parsed_numbers = self._execute_method(
7339:         "PolicyContradictionDetector", "_parse_number", context,
7340:         claims=quant_claims
7341:     )
7342:     significance_test = self._execute_method(
7343:         "PolicyContradictionDetector", "_statistical_significance_test", context,
7344:         numbers=parsed_numbers
7345:     )
7346:
7347:     # Step 6: Evaluate baseline quality and compare
7348:     metric_evaluation = self._execute_method(
7349:         "BayesianNumericalAnalyzer", "evaluate_policy_metric", context,
7350:         metrics=parsed_numbers
7351:     )
7352:     policy_comparison = self._execute_method(
7353:         "BayesianNumericalAnalyzer", "compare_policies", context,
7354:         evaluations=metric_evaluation
7355:     )
7356:
7357:     # Step 7: Semantic validation of sources
7358:     text_chunks = self._execute_method(
7359:         "SemanticProcessor", "chunk_text", context
7360:     )
7361:     embeddings = self._execute_method(
7362:         "SemanticProcessor", "embed_single", context,
7363:         chunks=text_chunks
7364:     )
7365:
7366:     # Assemble raw evidence
7367:     raw_evidence = {
7368:         "numeric_data": parsed_numbers,
7369:         "reference_years": [gc.get("year") for gc in goal_contexts if gc.get("year")],
7370:         "official_sources": point_evidence.get("sources", []),
7371:         "financial_baseline": financial_amounts,
7372:         "budget_tables": budget_table_data,
7373:         "significance_results": significance_test,
7374:         "metric_evaluation": metric_evaluation,
7375:         "source_embeddings": embeddings
7376:     }
7377:
7378:     return {
7379:         "executor_id": self.executor_id,
7380:         "raw_evidence": raw_evidence,
7381:         "metadata": {
7382:             "methods_executed": [log["method"] for log in self.execution_log],
7383:             "total_numeric_claims": len(parsed_numbers),
7384:             "sources_identified": len(point_evidence.get("sources", []))
7385:         },
7386:         "execution_metrics": {
7387:             "methods_count": len(self.execution_log),
```

```
7388:         "all_succeeded": all(log["success"] for log in self.execution_log)
7389:     }
7390: }
7391:
7392:
7393: class D1_Q2_ProblemDimensioningAnalyzer(BaseExecutor):
7394: """
7395:     Quantifies problem magnitude, gaps, and identifies data limitations.
7396:
7397:     Methods (from D1-Q2):
7398:     - OperationalizationAuditor._audit_direct_evidence
7399:     - OperationalizationAuditor._audit_systemic_risk
7400:     - FinancialAuditor._detect_allocation_gaps
7401:     - BayesianMechanismInference._detect_gaps
7402:     - PDET MunicipalPlanAnalyzer._generate_optimal_remediations
7403:     - PDET MunicipalPlanAnalyzer._simulate_intervention
7404:     - BayesianCounterfactualAuditor.counterfactual_query
7405:     - BayesianCounterfactualAuditor._test_effect_stability
7406:     - PolicyContradictionDetector._detect_numerical_inconsistencies
7407:     - PolicyContradictionDetector._calculate_numerical_divergence
7408:     - BayesianConfidenceCalculator.calculate_posterior
7409:     - PerformanceAnalyzer.analyze_performance
7410: """
7411:
7412: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7413:     raw_evidence = {}
7414:
7415:     # Step 1: Audit evidence completeness
7416:     direct_evidence_audit = self._execute_method(
7417:         "OperationalizationAuditor", "_audit_direct_evidence", context
7418:     )
7419:     systemic_risk_audit = self._execute_method(
7420:         "OperationalizationAuditor", "_audit_systemic_risk", context
7421:     )
7422:
7423:     # Step 2: Detect gaps in resource allocation and mechanisms
7424:     allocation_gaps = self._execute_method(
7425:         "FinancialAuditor", "_detect_allocation_gaps", context
7426:     )
7427:     mechanism_gaps = self._execute_method(
7428:         "BayesianMechanismInference", "_detect_gaps", context
7429:     )
7430:
7431:     # Step 3: Generate optimal remediations and simulate interventions
7432:     remediations = self._execute_method(
7433:         "PDET MunicipalPlanAnalyzer", "_generate_optimal_remediations", context,
7434:         gaps=allocation_gaps
7435:     )
7436:     simulation_results = self._execute_method(
7437:         "PDET MunicipalPlanAnalyzer", "_simulate_intervention", context,
7438:         remediations=remediations
7439:     )
7440:
7441:     # Step 4: Counterfactual analysis for problem dimensioning
7442:     counterfactual = self._execute_method(
7443:         "BayesianCounterfactualAuditor", "counterfactual_query", context
7444:
```

```
7444:         )
7445:         effect_stability = self._execute_method(
7446:             "BayesianCounterfactualAuditor", "_test_effect_stability", context,
7447:             counterfactual=counterfactual
7448:         )
7449:
7450:     # Step 5: Detect numerical inconsistencies
7451:     numerical_inconsistencies = self._execute_method(
7452:         "PolicyContradictionDetector", "_detect_numerical_inconsistencies", context
7453:     )
7454:     divergence_calc = self._execute_method(
7455:         "PolicyContradictionDetector", "_calculate_numerical_divergence", context,
7456:         inconsistencies=numerical_inconsistencies
7457:     )
7458:
7459:     # Step 6: Calculate confidence and analyze performance
7460:     posterior_confidence = self._execute_method(
7461:         "BayesianConfidenceCalculator", "calculate_posterior", context,
7462:         evidence=direct_evidence_audit
7463:     )
7464:     performance_analysis = self._execute_method(
7465:         "PerformanceAnalyzer", "analyze_performance", context
7466:     )
7467:
7468:     raw_evidence = {
7469:         "magnitude_indicators": {
7470:             "allocation_gaps": allocation_gaps,
7471:             "mechanism_gaps": mechanism_gaps,
7472:             "numerical_inconsistencies": numerical_inconsistencies
7473:         },
7474:         "deficit_quantification": divergence_calc,
7475:         "data_limitations": {
7476:             "evidence_gaps": direct_evidence_audit.get("gaps", []),
7477:             "systemic_risks": systemic_risk_audit
7478:         },
7479:         "simulation_results": simulation_results,
7480:         "confidence_scores": posterior_confidence,
7481:         "performance_metrics": performance_analysis
7482:     }
7483:
7484:     return {
7485:         "executor_id": self.executor_id,
7486:         "raw_evidence": raw_evidence,
7487:         "metadata": {
7488:             "methods_executed": [log["method"] for log in self.execution_log],
7489:             "gaps_identified": len(allocation_gaps) + len(mechanism_gaps),
7490:             "inconsistencies_found": len(numerical_inconsistencies)
7491:         },
7492:         "execution_metrics": {
7493:             "methods_count": len(self.execution_log),
7494:             "all_succeeded": all(log["success"] for log in self.execution_log)
7495:         }
7496:     }
7497:
7498:
7499: class D1_Q3_BudgetAllocationTracer(BaseExecutor):
```

```
7500: """
7501:     Traces monetary resources assigned to programs in Investment Plan (PPI).
7502:
7503:     Methods (from D1-Q3):
7504:         - FinancialAuditor.trace_financial_allocation
7505:         - FinancialAuditor._process_financial_table
7506:         - FinancialAuditor._match_program_to_node
7507:         - FinancialAuditor._match_goal_to_budget
7508:         - FinancialAuditor._perform_counterfactual_budget_check
7509:         - FinancialAuditor._calculate_sufficiency
7510:         - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility
7511:         - PDET MunicipalPlanAnalyzer._extract_budget_for_pillar
7512:         - PDET MunicipalPlanAnalyzer._identify_funding_source
7513:         - PDET MunicipalPlanAnalyzer._classify_tables
7514:         - PDET MunicipalPlanAnalyzer._analyze_funding_sources
7515:         - PDET MunicipalPlanAnalyzer._score_financial_component
7516:         - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
7517:
7518: """
7519: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7520:     raw_evidence = {}
7521:
7522:     # Step 1: Trace complete financial allocation chain
7523:     allocation_trace = self._execute_method(
7524:         "FinancialAuditor", "trace_financial_allocation", context
7525:     )
7526:     processed_tables = self._execute_method(
7527:         "FinancialAuditor", "_process_financial_table", context
7528:     )
7529:
7530:     # Step 2: Match programs to budget nodes
7531:     program_matches = self._execute_method(
7532:         "FinancialAuditor", "_match_program_to_node", context,
7533:         tables=processed_tables
7534:     )
7535:     goal_budget_matches = self._execute_method(
7536:         "FinancialAuditor", "_match_goal_to_budget", context,
7537:         programs=program_matches
7538:     )
7539:
7540:     # Step 3: Counterfactual checks and sufficiency calculation
7541:     counterfactual_check = self._execute_method(
7542:         "FinancialAuditor", "_perform_counterfactual_budget_check", context,
7543:         matches=goal_budget_matches
7544:     )
7545:     sufficiency_calc = self._execute_method(
7546:         "FinancialAuditor", "_calculate_sufficiency", context,
7547:         allocation=allocation_trace
7548:     )
7549:
7550:     # Step 4: Analyze financial feasibility
7551:     feasibility_analysis = self._execute_method(
7552:         "PDET MunicipalPlanAnalyzer", "analyze_financial_feasibility", context
7553:     )
7554:     pillar_budgets = self._execute_method(
7555:         "PDET MunicipalPlanAnalyzer", "_extract_budget_for_pillar", context
```

```
7556:         )
7557:         funding_sources = self._execute_method(
7558:             "PDETMunicipalPlanAnalyzer", "_identify_funding_source", context
7559:         )
7560:
7561:     # Step 5: Classify and analyze tables
7562:     table_classification = self._execute_method(
7563:         "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
7564:         tables=processed_tables
7565:     )
7566:     funding_analysis = self._execute_method(
7567:         "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
7568:         sources=funding_sources
7569:     )
7570:     financial_score = self._execute_method(
7571:         "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
7572:         analysis=funding_analysis
7573:     )
7574:
7575:     # Step 6: Aggregate risk and prioritize
7576:     risk_aggregation = self._execute_method(
7577:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
7578:         sufficiency=sufficiency_calc
7579:     )
7580:
7581:     raw_evidence = {
7582:         "budget_allocations": allocation_trace,
7583:         "program_mappings": program_matches,
7584:         "goal_budget_links": goal_budget_matches,
7585:         "sufficiency_analysis": sufficiency_calc,
7586:         "pillar_budgets": pillar_budgets,
7587:         "funding_sources": funding_sources,
7588:         "financial_feasibility": feasibility_analysis,
7589:         "financial_score": financial_score,
7590:         "risk_priorities": risk_aggregation
7591:     }
7592:
7593:     return {
7594:         "executor_id": self.executor_id,
7595:         "raw_evidence": raw_evidence,
7596:         "metadata": {
7597:             "methods_executed": [log["method"] for log in self.execution_log],
7598:             "programs_traced": len(program_matches),
7599:             "funding_sources_identified": len(funding_sources)
7600:         },
7601:         "execution_metrics": {
7602:             "methods_count": len(self.execution_log),
7603:             "all_succeeded": all(log["success"] for log in self.execution_log)
7604:         }
7605:     }
7606:
7607:
7608: class D1_Q4_InstitutionalCapacityIdentifier(BaseExecutor):
7609:     """
7610:         Identifies installed capacity (entities, staff, equipment) and limitations.
7611:
```

```
7612:     Methods (from D1-Q4):
7613:     - PDETMunicipalPlanAnalyzer.identify_responsible_entities
7614:     - PDETMunicipalPlanAnalyzer._extract_entities_ner
7615:     - PDETMunicipalPlanAnalyzer._extract_entities_syntax
7616:     - PDETMunicipalPlanAnalyzer._classify_entity_type
7617:     - PDETMunicipalPlanAnalyzer._score_entity_specificity
7618:     - PDETMunicipalPlanAnalyzer._consolidate_entities
7619:     - MechanismPartExtractor.extract_entity_activity
7620:     - MechanismPartExtractor._normalize_entity
7621:     - MechanismPartExtractor._validate_entity_activity
7622:     - MechanismPartExtractor._calculate_ea_confidence
7623:     - OperationalizationAuditor.audit_evidence_traceability
7624: """
7625:
7626: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7627:     raw_evidence = {}
7628:
7629:     # Step 1: Identify all responsible entities
7630:     entities_identified = self._execute_method(
7631:         "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
7632:     )
7633:
7634:     # Step 2: Extract entities using NER and syntax
7635:     ner_entities = self._execute_method(
7636:         "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context
7637:     )
7638:     syntax_entities = self._execute_method(
7639:         "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context
7640:     )
7641:
7642:     # Step 3: Classify and score entities
7643:     entity_types = self._execute_method(
7644:         "PDETMunicipalPlanAnalyzer", "_classify_entity_type", context,
7645:         entities=ner_entities + syntax_entities
7646:     )
7647:     specificity_scores = self._execute_method(
7648:         "PDETMunicipalPlanAnalyzer", "_score_entity_specificity", context,
7649:         entities=entity_types
7650:     )
7651:     consolidated = self._execute_method(
7652:         "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
7653:         entities=entity_types
7654:     )
7655:
7656:     # Step 4: Extract entity-activity relationships
7657:     entity_activities = self._execute_method(
7658:         "MechanismPartExtractor", "extract_entity_activity", context,
7659:         entities=consolidated
7660:     )
7661:     normalized = self._execute_method(
7662:         "MechanismPartExtractor", "_normalize_entity", context,
7663:         activities=entity_activities
7664:     )
7665:     validated = self._execute_method(
7666:         "MechanismPartExtractor", "_validate_entity_activity", context,
7667:         normalized=normalized
```

```
7668:         )
7669:         ea_confidence = self._execute_method(
7670:             "MechanismPartExtractor", "_calculate_ea_confidence", context,
7671:             validated=validated
7672:         )
7673:
7674:     # Step 5: Audit evidence traceability
7675:     traceability_audit = self._execute_method(
7676:         "OperationalizationAuditor", "audit_evidence_traceability", context,
7677:         entity_activities=validated
7678:     )
7679:
7680:     raw_evidence = {
7681:         "entities_identified": consolidated,
7682:         "entity_types": entity_types,
7683:         "specificity_scores": specificity_scores,
7684:         "entity_activities": validated,
7685:         "activity_confidence": ea_confidence,
7686:         "capacity_indicators": {
7687:             "staff_mentions": [e for e in consolidated if e.get("type") == "staff"],
7688:             "equipment_mentions": [e for e in consolidated if e.get("type") == "equipment"],
7689:             "organizational_units": [e for e in consolidated if e.get("type") == "organization"]
7690:         },
7691:         "limitations_identified": traceability_audit.get("gaps", []),
7692:         "traceability_audit": traceability_audit
7693:     }
7694:
7695:     return {
7696:         "executor_id": self.executor_id,
7697:         "raw_evidence": raw_evidence,
7698:         "metadata": {
7699:             "methods_executed": [log["method"] for log in self.execution_log],
7700:             "entities_count": len(consolidated),
7701:             "activities_extracted": len(validated)
7702:         },
7703:         "execution_metrics": {
7704:             "methods_count": len(self.execution_log),
7705:             "all_succeeded": all(log["success"] for log in self.execution_log)
7706:         }
7707:     }
7708:
7709:
7710: class D1_Q5_ScopeJustificationValidator(BaseExecutor):
7711:     """
7712:     Validates scope justification via legal framework and constraint recognition.
7713:
7714:     Methods (from D1-Q5):
7715:     - TemporalLogicVerifier._check_deadline_constraints
7716:     - TemporalLogicVerifier.verify_temporal_consistency
7717:     - CausalInferenceSetup.identify_failure_points
7718:     - CausalExtractor._assess_temporal_coherence
7719:     - TextMiningEngine._analyze_link_text
7720:     - IndustrialPolicyProcessor._analyze_causal_dimensions
7721:     - IndustrialPolicyProcessor._extract_metadata
7722:     """
7723:
```

```
7724:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7725:         raw_evidence = {}
7726:
7727:         # Step 1: Verify temporal constraints
7728:         deadline_constraints = self._execute_method(
7729:             "TemporalLogicVerifier", "_check_deadline_constraints", context
7730:         )
7731:         temporal_consistency = self._execute_method(
7732:             "TemporalLogicVerifier", "verify_temporal_consistency", context
7733:         )
7734:
7735:         # Step 2: Identify failure points in scope
7736:         failure_points = self._execute_method(
7737:             "CausalInferenceSetup", "identify_failure_points", context
7738:         )
7739:
7740:         # Step 3: Assess temporal coherence
7741:         temporal_coherence = self._execute_method(
7742:             "CausalExtractor", "assess_temporal_coherence", context
7743:         )
7744:
7745:         # Step 4: Analyze link text for justifications
7746:         link_analysis = self._execute_method(
7747:             "TextMiningEngine", "analyze_link_text", context
7748:         )
7749:
7750:         # Step 5: Analyze causal dimensions and extract metadata
7751:         causal_dimensions = self._execute_method(
7752:             "IndustrialPolicyProcessor", "analyze_causal_dimensions", context
7753:         )
7754:         metadata_extracted = self._execute_method(
7755:             "IndustrialPolicyProcessor", "extract_metadata", context,
7756:             dimensions=causal_dimensions
7757:         )
7758:
7759:         raw_evidence = {
7760:             "legal_framework_citations": metadata_extracted.get("legal_refs", []),
7761:             "temporal_constraints": {
7762:                 "deadline_checks": deadline_constraints,
7763:                 "consistency": temporal_consistency,
7764:                 "coherence": temporal_coherence
7765:             },
7766:             "budgetary_constraints": metadata_extracted.get("budget_limits", []),
7767:             "competence_constraints": metadata_extracted.get("competence_refs", []),
7768:             "failure_points": failure_points,
7769:             "scope_justifications": link_analysis.get("justifications", []),
7770:             "causal_dimensions": causal_dimensions
7771:         }
7772:
7773:         return {
7774:             "executor_id": self.executor_id,
7775:             "raw_evidence": raw_evidence,
7776:             "metadata": {
7777:                 "methods_executed": [log["method"] for log in self.execution_log],
7778:                 "constraints_identified": len(deadline_constraints),
7779:                 "legal_citations": len(metadata_extracted.get("legal_refs", []))}
```

```
7780:         },
7781:         "execution_metrics": {
7782:             "methods_count": len(self.execution_log),
7783:             "all_succeeded": all(log["success"] for log in self.execution_log)
7784:         }
7785:     }
7786:
7787:
7788: # =====
7789: # DIMENSION 2: ACTIVITY DESIGN
7790: # =====
7791:
7792: class D2_Q1_StructuredPlanningValidator(BaseExecutor):
7793:     """
7794:     Validates structured format of activities (table/matrix with required columns).
7795:
7796:     Methods (from D2-Q1):
7797:     - PDFProcessor.extract_tables
7798:     - FinancialAuditor._process_financial_table
7799:     - PDETMunicipalPlanAnalyzer._deduplicate_tables
7800:     - PDETMunicipalPlanAnalyzer._classify_tables
7801:     - PDETMunicipalPlanAnalyzer._is_likely_header
7802:     - PDETMunicipalPlanAnalyzer._clean_dataframe
7803:     - ReportingEngine.generate_accountability_matrix
7804:     """
7805:
7806:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7807:         raw_evidence = {}
7808:
7809:         # Step 1: Extract all tables
7810:         extracted_tables = self._execute_method(
7811:             "PDFProcessor", "extract_tables", context
7812:         )
7813:
7814:         # Step 2: Process financial tables
7815:         processed_tables = self._execute_method(
7816:             "FinancialAuditor", "_process_financial_table", context,
7817:             tables=extracted_tables
7818:         )
7819:
7820:         # Step 3: Deduplicate and classify tables
7821:         deduplicated = self._execute_method(
7822:             "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
7823:             tables=processed_tables
7824:         )
7825:         classified = self._execute_method(
7826:             "PDETMunicipalPlanAnalyzer", "_classify_tables", context,
7827:             tables=deduplicated
7828:         )
7829:
7830:         # Step 4: Identify headers and clean dataframes
7831:         header_checks = self._execute_method(
7832:             "PDETMunicipalPlanAnalyzer", "_is_likely_header", context,
7833:             tables=classified
7834:         )
7835:         cleaned = self._execute_method(
```

```

7836:         "PDET Municipal Plan Analyzer", "_clean_dataframe", context,
7837:         tables=classified
7838:     )
7839:
7840:     # Step 5: Generate accountability matrix
7841:     accountability_matrix = self._execute_method(
7842:         "ReportingEngine", "generate_accountability_matrix", context,
7843:         tables=cleaned
7844:     )
7845:
7846:     raw_evidence = {
7847:         "tables_extracted": len(extracted_tables),
7848:         "activity_tables": [t for t in classified if t.get("type") == "activity"],
7849:         "matrix_structure": accountability_matrix,
7850:         "required_columns_present": {
7851:             "responsible_entity": any("responsible" in str(t.get("columns", [])).lower()
7852:                                         for t in cleaned),
7853:             "deliverable": any("deliverable" in str(t.get("columns", [])).lower()
7854:                                for t in cleaned),
7855:             "timeline": any("timeline" in str(t.get("columns", [])).lower()
7856:                            for t in cleaned),
7857:             "cost": any("cost" in str(t.get("columns", [])).lower()
7858:                         for t in cleaned)
7859:         },
7860:         "table_quality": {
7861:             "clean_tables": len(cleaned),
7862:             "with_headers": sum(1 for h in header_checks if h)
7863:         }
7864:     }
7865:
7866:     return {
7867:         "executor_id": self.executor_id,
7868:         "raw_evidence": raw_evidence,
7869:         "metadata": {
7870:             "methods_executed": [log["method"] for log in self.execution_log],
7871:             "total_tables": len(extracted_tables),
7872:             "activity_tables": len([t for t in classified if t.get("type") == "activity"])
7873:         },
7874:         "execution_metrics": {
7875:             "methods_count": len(self.execution_log),
7876:             "all_succeeded": all(log["success"] for log in self.execution_log)
7877:         }
7878:     }
7879:
7880:
7881: class D2_Q2_InterventionLogicInferencer(BaseExecutor):
7882:     """
7883:         Infers intervention logic: instrument (how), target (who), causality (why).
7884:
7885:     Methods (from D2-Q2):
7886:     - BayesianMechanismInference.infer_mechanisms
7887:     - BayesianMechanismInference._infer_single_mechanism
7888:     - BayesianMechanismInference._infer_mechanism_type
7889:     - BayesianMechanismInference._test_sufficiency
7890:     - BayesianMechanismInference._test_necessity
7891:     - CausalExtractor.extract_causal_hierarchy

```

```
7892:     - TeoriaCambio.construir_grafo_causal
7893:     - TeoriaCambio._esConexionValida
7894:     - PDETmunicipalPlanAnalyzer.construct_causal_dag
7895:     - BeachEvidentialTest.classify_test
7896:     - IndustrialPolicyProcessor._analyze_causal_dimensions
7897: """
7898:
7899: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
7900:     raw_evidence = {}
7901:
7902:     # Step 1: Infer mechanisms
7903:     mechanisms = self._execute_method(
7904:         "BayesianMechanismInference", "infer_mechanisms", context
7905:     )
7906:     single_mechanisms = []
7907:     for mech in mechanisms:
7908:         single = self._execute_method(
7909:             "BayesianMechanismInference", "_infer_single_mechanism", context,
7910:             mechanism=mech
7911:         )
7912:         single_mechanisms.append(single)
7913:
7914:     mechanism_types = self._execute_method(
7915:         "BayesianMechanismInference", "_infer_mechanism_type", context,
7916:         mechanisms=single_mechanisms
7917:     )
7918:
7919:     # Step 2: Test sufficiency and necessity
7920:     sufficiency_tests = self._execute_method(
7921:         "BayesianMechanismInference", "_test_sufficiency", context,
7922:         mechanisms=single_mechanisms
7923:     )
7924:     necessity_tests = self._execute_method(
7925:         "BayesianMechanismInference", "_test_necessity", context,
7926:         mechanisms=single_mechanisms
7927:     )
7928:
7929:     # Step 3: Extract causal hierarchy
7930:     causal_hierarchy = self._execute_method(
7931:         "CausalExtractor", "extract_causal_hierarchy", context
7932:     )
7933:
7934:     # Step 4: Build causal graph
7935:     causal_graph = self._execute_method(
7936:         "TeoriaCambio", "construir_grafo_causal", context,
7937:         hierarchy=causal_hierarchy
7938:     )
7939:     connection_validation = self._execute_method(
7940:         "TeoriaCambio", "_esConexionValida", context,
7941:         graph=causal_graph
7942:     )
7943:
7944:     # Step 5: Construct DAG
7945:     causal_dag = self._execute_method(
7946:         "PDETmunicipalPlanAnalyzer", "construct_causal_dag", context,
7947:         graph=causal_graph
```

```
7948:     )
7949:
7950:     # Step 6: Classify evidential tests
7951:     evidential_tests = self._execute_method(
7952:         "BeachEvidentialTest", "classify_test", context,
7953:         mechanisms=single_mechanisms
7954:     )
7955:
7956:     # Step 7: Analyze causal dimensions
7957:     causal_dimensions = self._execute_method(
7958:         "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
7959:     )
7960:
7961:     raw_evidence = {
7962:         "intervention_instruments": [m.get("instrument") for m in single_mechanisms],
7963:         "target_populations": [m.get("target") for m in single_mechanisms],
7964:         "causal_logic": {
7965:             "mechanisms": single_mechanisms,
7966:             "mechanism_types": mechanism_types,
7967:             "sufficiency": sufficiency_tests,
7968:             "necessity": necessity_tests
7969:         },
7970:         "causal_hierarchy": causal_hierarchy,
7971:         "causal_graph": causal_graph,
7972:         "causal_dag": causal_dag,
7973:         "evidential_strength": evidential_tests,
7974:         "dimensions": causal_dimensions
7975:     }
7976:
7977:     return {
7978:         "executor_id": self.executor_id,
7979:         "raw_evidence": raw_evidence,
7980:         "metadata": {
7981:             "methods_executed": [log["method"] for log in self.execution_log],
7982:             "mechanisms_identified": len(single_mechanisms),
7983:             "instruments_found": len([m for m in single_mechanisms if m.get("instrument")])
7984:         },
7985:         "execution_metrics": {
7986:             "methods_count": len(self.execution_log),
7987:             "all_succeeded": all(log["success"] for log in self.execution_log)
7988:         }
7989:     }
7990:
7991:
7992: class D2_Q3_RootCauseLinkageAnalyzer(BaseExecutor):
7993:     """
7994:     Analyzes linkage between activities and root causes/structural determinants.
7995:
7996:     Methods (from D2-Q3):
7997:     - CausalExtractor._extract_causal_links
7998:     - CausalExtractor._calculate_composite_likelihood
7999:     - CausalExtractor._initialize_prior
8000:     - CausalExtractor._calculate_type_transition_prior
8001:     - PDET MunicipalPlanAnalyzer._identify_causal_edges
8002:     - PDET MunicipalPlanAnalyzer._refine_edge_probabilities
8003:     - BayesianCounterfactualAuditor.construct_scm
```

```
8004:     - BayesianCounterfactualAuditor._create_default_equations
8005:     - SemanticAnalyzer.extract_semantic_cube
8006:     """
8007:
8008:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8009:         raw_evidence = {}
8010:
8011:         # Step 1: Extract causal links
8012:         causal_links = self._execute_method(
8013:             "CausalExtractor", "_extract_causal_links", context
8014:         )
8015:
8016:         # Step 2: Calculate likelihoods
8017:         composite_likelihood = self._execute_method(
8018:             "CausalExtractor", "_calculate_composite_likelihood", context,
8019:             links=causal_links
8020:         )
8021:         prior_init = self._execute_method(
8022:             "CausalExtractor", "_initialize_prior", context
8023:         )
8024:         type_transition_prior = self._execute_method(
8025:             "CausalExtractor", "_calculate_type_transition_prior", context,
8026:             links=causal_links
8027:         )
8028:
8029:         # Step 3: Identify and refine causal edges
8030:         causal_edges = self._execute_method(
8031:             "PDET Municipal Plan Analyzer", "_identify_causal_edges", context,
8032:             links=causal_links
8033:         )
8034:         refined_probabilities = self._execute_method(
8035:             "PDET Municipal Plan Analyzer", "_refine_edge_probabilities", context,
8036:             edges=causal_edges
8037:         )
8038:
8039:         # Step 4: Construct structural causal model
8040:         scm = self._execute_method(
8041:             "BayesianCounterfactualAuditor", "construct_scm", context,
8042:             edges=refined_probabilities
8043:         )
8044:         default_equations = self._execute_method(
8045:             "BayesianCounterfactualAuditor", "_create_default_equations", context,
8046:             scm=scm
8047:         )
8048:
8049:         # Step 5: Extract semantic cube
8050:         semantic_cube = self._execute_method(
8051:             "SemanticAnalyzer", "extract_semantic_cube", context
8052:         )
8053:
8054:         raw_evidence = {
8055:             "root_causes_identified": [link.get("root_cause") for link in causal_links],
8056:             "activity_linkages": causal_links,
8057:             "link_probabilities": refined_probabilities,
8058:             "composite_likelihood": composite_likelihood,
8059:             "structural_model": scm,
```

```
8060:         "model_equations": default_equations,
8061:         "semantic_relationships": semantic_cube,
8062:         "determinants_addressed": [link for link in causal_links if link.get("addresses_determinant")]
8063:     }
8064:
8065:     return {
8066:         "executor_id": self.executor_id,
8067:         "raw_evidence": raw_evidence,
8068:         "metadata": {
8069:             "methods_executed": [log["method"] for log in self.execution_log],
8070:             "causal_links_found": len(causal_links),
8071:             "root_causes_count": len(set(link.get("root_cause") for link in causal_links))
8072:         },
8073:         "execution_metrics": {
8074:             "methods_count": len(self.execution_log),
8075:             "all_succeeded": all(log["success"] for log in self.execution_log)
8076:         }
8077:     }
8078:
8079:
8080: class D2_Q4_RiskManagementAnalyzer(BaseExecutor):
8081:     """
8082:         Identifies implementation risks and mitigation measures.
8083:
8084:         Methods (from D2-Q4):
8085:         - PDETMunicipalPlanAnalyzer._bayesian_risk_inference
8086:         - PDETMunicipalPlanAnalyzer.sensitivity_analysis
8087:         - PDETMunicipalPlanAnalyzer._interpret_risk
8088:         - PDETMunicipalPlanAnalyzer._compute_robustness_value
8089:         - PDETMunicipalPlanAnalyzer._compute_e_value
8090:         - PDETMunicipalPlanAnalyzer._interpret_sensitivity
8091:         - OperationalizationAuditor._audit_systemic_risk
8092:         - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
8093:         - BayesianCounterfactualAuditor.refutation_and_sanity_checks
8094:         - AdaptivePriorCalculator.sensitivity_analysis
8095:     """
8096:
8097:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8098:         raw_evidence = {}
8099:
8100:         # Step 1: Bayesian risk inference
8101:         risk_inference = self._execute_method(
8102:             "PDETMunicipalPlanAnalyzer", "_bayesian_risk_inference", context
8103:         )
8104:
8105:         # Step 2: Sensitivity analysis
8106:         sensitivity = self._execute_method(
8107:             "PDETMunicipalPlanAnalyzer", "sensitivity_analysis", context,
8108:             risks=risk_inference
8109:         )
8110:
8111:         # Step 3: Risk interpretation
8112:         risk_interpretation = self._execute_method(
8113:             "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
8114:             inference=risk_inference
8115:         )
```

```
8116:  
8117:      # Step 4: Compute robustness metrics  
8118:      robustness = self._execute_method(  
8119:          "PDET MunicipalPlanAnalyzer", "_compute_robustness_value", context,  
8120:          sensitivity=sensitivity  
8121:      )  
8122:      e_value = self._execute_method(  
8123:          "PDET MunicipalPlanAnalyzer", "_compute_e_value", context,  
8124:          robustness=robustness  
8125:      )  
8126:      sensitivity_interpretation = self._execute_method(  
8127:          "PDET MunicipalPlanAnalyzer", "_interpret_sensitivity", context,  
8128:          sensitivity=sensitivity  
8129:      )  
8130:  
8131:      # Step 5: Audit systemic risks  
8132:      systemic_risk_audit = self._execute_method(  
8133:          "OperationalizationAuditor", "_audit_systemic_risk", context  
8134:      )  
8135:  
8136:      # Step 6: Aggregate and prioritize risks  
8137:      risk_aggregation = self._execute_method(  
8138:          "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,  
8139:          risks=risk_inference  
8140:      )  
8141:  
8142:      # Step 7: Refutation and sanity checks  
8143:      refutation_checks = self._execute_method(  
8144:          "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,  
8145:          aggregation=risk_aggregation  
8146:      )  
8147:  
8148:      # Step 8: Additional sensitivity analysis  
8149:      adaptive_sensitivity = self._execute_method(  
8150:          "AdaptivePriorCalculator", "sensitivity_analysis", context,  
8151:          risks=risk_inference  
8152:      )  
8153:  
8154:      raw_evidence = {  
8155:          "operational_risks": [r for r in risk_inference if r.get("type") == "operational"],  
8156:          "social_risks": [r for r in risk_inference if r.get("type") == "social"],  
8157:          "security_risks": [r for r in risk_inference if r.get("type") == "security"],  
8158:          "mitigation_measures": risk_interpretation.get("mitigations", []),  
8159:          "risk_priorities": risk_aggregation,  
8160:          "robustness_metrics": {  
8161:              "robustness_value": robustness,  
8162:              "e_value": e_value  
8163:          },  
8164:          "sensitivity_analysis": sensitivity,  
8165:          "systemic_risks": systemic_risk_audit,  
8166:          "validation_checks": refutation_checks  
8167:      }  
8168:  
8169:      return {  
8170:          "executor_id": self.executor_id,  
8171:          "raw_evidence": raw_evidence,
```

```
8172:         "metadata": {
8173:             "methods_executed": [log["method"] for log in self.execution_log],
8174:             "risks_identified": len(risk_inference),
8175:             "mitigations_proposed": len(risk_interpretation.get("mitigations", []))
8176:         },
8177:         "execution_metrics": {
8178:             "methods_count": len(self.execution_log),
8179:             "all_succeeded": all(log["success"] for log in self.execution_log)
8180:         }
8181:     }
8182:
8183:
8184: class D2_Q5_StrategicCoherenceEvaluator(BaseExecutor):
8185:     """
8186:         Evaluates strategic coherence: complementarity and logical sequence.
8187:
8188:     Methods (from D2-Q5):
8189:     - PolicyContradictionDetector._detect_logical_incompatibilities
8190:     - PolicyContradictionDetector._calculate_coherence_metrics
8191:     - PolicyContradictionDetector._calculate_objective_alignment
8192:     - PolicyContradictionDetector._calculate_graph_fragmentation
8193:     - OperationalizationAuditor.audit_sequence_logic
8194:     - BayesianMechanismInference._calculate_coherence_factor
8195:     - PDET MunicipalPlanAnalyzer._score_causal_coherence
8196:     - AdaptivePriorCalculator.calculate_likelihood_adaptativo
8197:     """
8198:
8199:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8200:         raw_evidence = {}
8201:
8202:         # Step 1: Detect logical incompatibilities
8203:         incompatibilities = self._execute_method(
8204:             "PolicyContradictionDetector", "_detect_logical_incompatibilities", context
8205:         )
8206:
8207:         # Step 2: Calculate coherence metrics
8208:         coherence_metrics = self._execute_method(
8209:             "PolicyContradictionDetector", "_calculate_coherence_metrics", context
8210:         )
8211:         objective_alignment = self._execute_method(
8212:             "PolicyContradictionDetector", "_calculate_objective_alignment", context
8213:         )
8214:         graph_fragmentation = self._execute_method(
8215:             "PolicyContradictionDetector", "_calculate_graph_fragmentation", context
8216:         )
8217:
8218:         # Step 3: Audit sequence logic
8219:         sequence_audit = self._execute_method(
8220:             "OperationalizationAuditor", "audit_sequence_logic", context
8221:         )
8222:
8223:         # Step 4: Calculate coherence factors
8224:         coherence_factor = self._execute_method(
8225:             "BayesianMechanismInference", "_calculate_coherence_factor", context,
8226:             metrics=coherence_metrics
8227:         )
```

```

8228:     causal_coherence_score = self._execute_method(
8229:         "PDET MunicipalPlanAnalyzer", "_score_causal_coherence", context
8230:     )
8231:
8232:     # Step 5: Adaptive likelihood calculation
8233:     adaptive_likelihood = self._execute_method(
8234:         "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context,
8235:         coherence=causal_coherence_score
8236:     )
8237:
8238:     raw_evidence = {
8239:         "complementarity_evidence": coherence_metrics.get("complementarity", []),
8240:         "sequential_logic": sequence_audit,
8241:         "logical_incompatibilities": incompatibilities,
8242:         "coherence_scores": {
8243:             "overall_coherence": coherence_metrics,
8244:             "objective_alignment": objective_alignment,
8245:             "causal_coherence": causal_coherence_score,
8246:             "coherence_factor": coherence_factor
8247:         },
8248:         "graph_metrics": {
8249:             "fragmentation": graph_fragmentation
8250:         },
8251:         "adaptive_likelihood": adaptive_likelihood
8252:     }
8253:
8254:     return {
8255:         "executor_id": self.executor_id,
8256:         "raw_evidence": raw_evidence,
8257:         "metadata": {
8258:             "methods_executed": [log["method"] for log in self.execution_log],
8259:             "incompatibilities_found": len(incompatibilities),
8260:             "coherence_score": coherence_metrics.get("score", 0)
8261:         },
8262:         "execution_metrics": {
8263:             "methods_count": len(self.execution_log),
8264:             "all_succeeded": all(log["success"] for log in self.execution_log)
8265:         }
8266:     }
8267:
8268:
8269: # =====
8270: # DIMENSION 3: PRODUCTS & OUTPUTS
8271: # =====
8272:
8273: class D3_Q1_IndicatorQualityValidator(BaseExecutor):
8274:     """
8275:         Validates indicator quality: baseline, target, source of verification.
8276:
8277:         Methods (from D3-Q1):
8278:         - PDET MunicipalPlanAnalyzer._score_indicators
8279:         - OperationalizationAuditor.audit_evidence_traceability
8280:         - CausalInferenceSetup.assign_probative_value
8281:         - BeachEvidentialTest.apply_test_logic
8282:         - TextMiningEngine.diagnose_critical_links
8283:         - IndustrialPolicyProcessor._extract_metadata

```

```
8284:     - IndustrialPolicyProcessor._calculate_quality_score
8285:     - AdaptivePriorCalculator.generate_traceability_record
8286:     """
8287:
8288:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8289:         raw_evidence = {}
8290:
8291:         # Step 1: Score indicators
8292:         indicator_scores = self._execute_method(
8293:             "PDET MunicipalPlanAnalyzer", "_score_indicators", context
8294:         )
8295:
8296:         # Step 2: Audit evidence traceability
8297:         traceability_audit = self._execute_method(
8298:             "OperationalizationAuditor", "audit_evidence_traceability", context,
8299:             indicators=indicator_scores
8300:         )
8301:
8302:         # Step 3: Assign probative value
8303:         probative_values = self._execute_method(
8304:             "CausalInferenceSetup", "assign_probative_value", context,
8305:             indicators=indicator_scores
8306:         )
8307:
8308:         # Step 4: Apply evidential tests
8309:         evidential_tests = self._execute_method(
8310:             "BeachEvidentialTest", "apply_test_logic", context,
8311:             indicators=indicator_scores
8312:         )
8313:
8314:         # Step 5: Diagnose critical links
8315:         critical_links = self._execute_method(
8316:             "TextMiningEngine", "diagnose_critical_links", context
8317:         )
8318:
8319:         # Step 6: Extract and score metadata
8320:         metadata = self._execute_method(
8321:             "IndustrialPolicyProcessor", "_extract_metadata", context
8322:         )
8323:         quality_score = self._execute_method(
8324:             "IndustrialPolicyProcessor", "_calculate_quality_score", context,
8325:             metadata=metadata
8326:         )
8327:
8328:         # Step 7: Generate traceability record
8329:         traceability_record = self._execute_method(
8330:             "AdaptivePriorCalculator", "generate_traceability_record", context,
8331:             indicators=indicator_scores
8332:         )
8333:
8334:         raw_evidence = {
8335:             "indicators_with_baseline": [i for i in indicator_scores if i.get("has_baseline")],
8336:             "indicators_with_target": [i for i in indicator_scores if i.get("has_target")],
8337:             "indicators_with_source": [i for i in indicator_scores if i.get("has_source")],
8338:             "indicator_quality_scores": indicator_scores,
8339:             "traceability": traceability_audit,
```

```

8340:         "probative_values": probative_values,
8341:         "evidential_strength": evidential_tests,
8342:         "overall_quality_score": quality_score,
8343:         "traceability_record": traceability_record
8344:     }
8345:
8346:     return {
8347:         "executor_id": self.executor_id,
8348:         "raw_evidence": raw_evidence,
8349:         "metadata": {
8350:             "methods_executed": [log["method"] for log in self.execution_log],
8351:             "total_indicators": len(indicator_scores),
8352:             "complete_indicators": len([i for i in indicator_scores
8353:                 if i.get("has_baseline") and i.get("has_target") and i.get("has_source")])
8354:         },
8355:         "execution_metrics": {
8356:             "methods_count": len(self.execution_log),
8357:             "all_succeeded": all(log["success"] for log in self.execution_log)
8358:         }
8359:     }
8360:
8361:
8362: class D3_Q2_TargetProportionalityAnalyzer(BaseExecutor):
8363:     """
8364:         DIM03_Q02_PRODUCT_TARGET_PROPORIONALITY \200\224 Analyzes proportionality of targets to the diagnosed universe using canonical D3 notation.
8365:         Epistemic mix: structural coverage, financial/normative feasibility, statistical Bayes tests, and semantic indicator quality.
8366:
8367:         Methods (from D3-Q2):
8368:         - AdvancedDAGValidator._calculate_bayesian_posterior
8369:         - AdvancedDAGValidator._calculate_confidence_interval
8370:         - AdaptivePriorCalculator._adjust_domain_weights
8371:         - PDET MunicipalPlanAnalyzer._get_spanish_stopwords
8372:         - BayesianMechanismInference._log_refactored_components
8373:         - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility
8374:         - PDET MunicipalPlanAnalyzer._score_indicators
8375:         - PDET MunicipalPlanAnalyzer._interpret_risk
8376:         - FinancialAuditor._calculate_sufficiency
8377:         - BayesianMechanismInference._test_sufficiency
8378:         - BayesianMechanismInference._test_necessity
8379:         - PDET MunicipalPlanAnalyzer._assess_financial_sustainability
8380:         - AdaptivePriorCalculator.calculate_likelihood_adaptativo
8381:         - IndustrialPolicyProcessor._calculate_quality_score
8382:         - TeoriaCambio._generar_sugerencias_internas
8383:         - PDET MunicipalPlanAnalyzer._deduplicate_tables
8384:         - PDET MunicipalPlanAnalyzer._indicator_to_dict
8385:         - PDET MunicipalPlanAnalyzer._generate_recommendations
8386:         - IndustrialPolicyProcessor._compile_pattern_registry
8387:         - IndustrialPolicyProcessor._build_point_patterns
8388:         - IndustrialPolicyProcessor._empty_result
8389:     """
8390:
8391:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8392:         raw_evidence = {}
8393:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
8394:
8395:         # Step 0: Financial feasibility snapshot and indicator quality

```

```
8396:     financial_feasibility = self._execute_method(
8397:         "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
8398:     )
8399:     indicator_quality = self._execute_method(
8400:         "PDETMunicipalPlanAnalyzer", "_score_indicators", context
8401:     )
8402:     spanish_stopwords = self._execute_method(
8403:         "PDETMunicipalPlanAnalyzer", "_get_spanish_stopwords", context
8404:     )
8405:     funding_sources = self._execute_method(
8406:         "PDETMunicipalPlanAnalyzer", "_analyze_funding_sources", context,
8407:         financial_indicators=financial_feasibility.get("financial_indicators", []),
8408:         tables=context.get("tables", [])
8409:     )
8410:     financial_component = self._execute_method(
8411:         "PDETMunicipalPlanAnalyzer", "_score_financial_component", context,
8412:         financial_analysis=financial_feasibility
8413:     )
8414:     pattern_registry = self._execute_method(
8415:         "IndustrialPolicyProcessor", "_compile_pattern_registry", context
8416:     )
8417:     point_patterns = self._execute_method(
8418:         "IndustrialPolicyProcessor", "_build_point_patterns", context
8419:     )
8420:     empty_policy_result = self._execute_method(
8421:         "IndustrialPolicyProcessor", "_empty_result", context
8422:     )
8423:     dedup_tables = self._execute_method(
8424:         "PDETMunicipalPlanAnalyzer", "_deduplicate_tables", context,
8425:         tables=context.get("tables", [])
8426:     )
8427:     first_indicator = None
8428:     if isinstance(financial_feasibility.get("financial_indicators", []), list):
8429:         inds = financial_feasibility.get("financial_indicators", [])
8430:         first_indicator = inds[0] if inds else None
8431:     indicator_dict = self._execute_method(
8432:         "PDETMunicipalPlanAnalyzer", "_indicator_to_dict", context,
8433:         ind=first_indicator if first_indicator else {}
8434:     )
8435:     proportionality_recommendations = self._execute_method(
8436:         "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
8437:         analysis_results={"financial_analysis": financial_feasibility, "quality_score": quality_score} if 'quality_score' in locals() else {}
8438:     )
8439:
8440:     # Step 1: Calculate sufficiency
8441:     sufficiency_calc = self._execute_method(
8442:         "FinancialAuditor", "_calculate_sufficiency", context
8443:     )
8444:
8445:     # Step 2: Test sufficiency and necessity of targets
8446:     sufficiency_test = self._execute_method(
8447:         "BayesianMechanismInference", "_test_sufficiency", context
8448:     )
8449:     necessity_test = self._execute_method(
8450:         "BayesianMechanismInference", "_test_necessity", context
8451:     )
```

```
8452:  
8453:     # Step 3: Assess financial sustainability  
8454:     sustainability_assessment = self._execute_method(  
8455:         "PDET MunicipalPlanAnalyzer", "_assess_financial_sustainability", context  
8456:     )  
8457:     risk_interpretation = self._execute_method(  
8458:         "PDET MunicipalPlanAnalyzer", "_interpret_risk", context,  
8459:         risk=financial_feasibility.get("risk_assessment", {}).get("risk_score", 0.0)  
8460:     )  
8461:  
8462:     # Step 4: Calculate adaptive likelihood  
8463:     adaptive_likelihood = self._execute_method(  
8464:         "AdaptivePriorCalculator", "calculate_likelihood_adaptativo", context  
8465:     )  
8466:     domain_scores = {  
8467:         "structural": sufficiency_calc.get("coverage_ratio", 0.0),  
8468:         "financial": financial_feasibility.get("sustainability_score", 0.0),  
8469:         "semantic": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0  
8470:     }  
8471:     adjusted_weights = self._execute_method(  
8472:         "AdaptivePriorCalculator", "_adjust_domain_weights", context,  
8473:         domain_scores=domain_scores  
8474:     )  
8475:     avg_confidence = self._execute_method(  
8476:         "IndustrialPolicyProcessor", "_compute_avg_confidence", context,  
8477:         dimension_analysis={"D3": {"dimension_confidence": domain_scores.get("structural", 0.0)}}  
8478:     )  
8479:  
8480:     # Step 5: Calculate quality score  
8481:     quality_score = self._execute_method(  
8482:         "IndustrialPolicyProcessor", "_calculate_quality_score", context  
8483:     )  
8484:  
8485:     # Step 6: Generate internal suggestions  
8486:     internalSuggestions = self._execute_method(  
8487:         "TeoriaCambio", "_generar_sugerencias_internas", context  
8488:     )  
8489:     # Bayesian posterior diagnostics for proportionality evidence  
8490:     posterior_probability = self._execute_method(  
8491:         "AdvancedDAGValidator", "_calculate_bayesian_posterior", context,  
8492:         likelihood=sufficiency_calc.get("coverage_ratio", 0.5),  
8493:         prior=0.5  
8494:     )  
8495:     confidence_interval = self._execute_method(  
8496:         "AdvancedDAGValidator", "_calculate_confidence_interval", context,  
8497:         s=int(sufficiency_calc.get("covered_targets", 0)),  
8498:         n=max(1, int(sufficiency_calc.get("targets_total", len(context.get("product_targets", [])))),  
8499:         conf=0.95  
8500:     )  
8501:     self._execute_method(  
8502:         "BayesianMechanismInference", "_log_refactored_components", context  
8503:     )  
8504:  
8505:     raw_evidence = {  
8506:         "target_population_size": context.get("diagnosed_universe", 0),  
8507:         "product_targets": context.get("product_targets", []),
```

```
8508:         "coverage_ratio": sufficiency_calc.get("coverage_ratio", 0),
8509:         "dosage_analysis": sufficiency_calc.get("dosage", {}),
8510:         "sufficiency_test": sufficiency_test,
8511:         "necessity_test": necessity_test,
8512:         "sustainability": sustainability_assessment,
8513:         "financial_feasibility": financial_feasibility,
8514:         "indicator_quality": indicator_quality,
8515:         "risk_interpretation": risk_interpretation,
8516:         "proportionality_score": quality_score,
8517:         "recommendations": internal_suggestions,
8518:         "stopwords_spanish": spanish_stopwords,
8519:         "funding_sources_analysis": funding_sources,
8520:         "financial_component_score": financial_component,
8521:         "pattern_registry": pattern_registry,
8522:         "point_patterns": point_patterns,
8523:         "empty_policy_result": empty_policy_result,
8524:         "avg_confidence": avg_confidence,
8525:         "deduplicated_tables": dedup_tables,
8526:         "indicator_sample": indicator_dict,
8527:         "proportionality_recommendations": proportionality_recommendations,
8528:         "adjusted_domain_weights": adjusted_weights,
8529:         "posterior_proportionality": posterior_probability,
8530:         "coverage_interval": confidence_interval
8531:     }
8532:
8533:     return {
8534:         "executor_id": self.executor_id,
8535:         "raw_evidence": raw_evidence,
8536:         "metadata": {
8537:             "methods_executed": [log["method"] for log in self.execution_log],
8538:             "targets_analyzed": len(context.get("product_targets", [])),
8539:             "coverage_adequate": sufficiency_calc.get("is_sufficient", False),
8540:             "canonical_question": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
8541:             "dimension_code": dim_info.code,
8542:             "dimension_label": dim_info.label
8543:         },
8544:         "execution_metrics": {
8545:             "methods_count": len(self.execution_log),
8546:             "all_succeeded": all(log["success"] for log in self.execution_log)
8547:         }
8548:     }
8549:
8550:
8551: class D3_Q3_TraceabilityValidator(BaseExecutor):
8552:     """
8553:     DIM03_Q03_TRACEABILITY_BUDGET_ORG \200\224 Validates budgetary and organizational traceability of products under canonical D3 notation.
8554:     Epistemic mix: structural budget tracing, organizational semantics, and accountability synthesis.
8555:
8556:     Methods (from D3-Q3):
8557:     - PolicyAnalysisEmbedder.process_document
8558:     - PolicyAnalysisEmbedder.semantic_search
8559:     - PolicyAnalysisEmbedder._apply_mmr
8560:     - PolicyAnalysisEmbedder._generate_query_from_pdq
8561:     - SemanticAnalyzer._empty_semantic_cube
8562:     - FinancialAuditor._match_program_to_node
8563:     - FinancialAuditor._match_goal_to_budget
```

```
8564:     - PDETMunicipalPlanAnalyzer._extract_from_responsibility_tables
8565:     - PDETMunicipalPlanAnalyzer._consolidate_entities
8566:     - AdaptivePriorCalculator.generate_traceability_record
8567:     - PolicyAnalysisEmbedder.generate_pdq_report
8568:     - ReportingEngine.generate_accountability_matrix
8569:     """
8570:
8571:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8572:         raw_evidence = {}
8573:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
8574:         document_text = context.get("document_text", "")
8575:         document_metadata = context.get("metadata", {})
8576:
8577:         # Step 1: Match programs to budget nodes
8578:         program_matches = self._execute_method(
8579:             "FinancialAuditor", "_match_program_to_node", context
8580:         )
8581:         goal_budget_matches = self._execute_method(
8582:             "FinancialAuditor", "_match_goal_to_budget", context,
8583:             programs=program_matches
8584:         )
8585:
8586:         # Step 2: Extract responsibility assignments
8587:         responsibility_data = self._execute_method(
8588:             "PDETMunicipalPlanAnalyzer", "_extract_from_responsibility_tables", context
8589:         )
8590:         consolidated_entities = self._execute_method(
8591:             "PDETMunicipalPlanAnalyzer", "_consolidate_entities", context,
8592:             entities=responsibility_data
8593:         )
8594:         responsible_entities = self._execute_method(
8595:             "PDETMunicipalPlanAnalyzer", "identify_responsible_entities", context
8596:         )
8597:         responsibility_clarity = self._execute_method(
8598:             "PDETMunicipalPlanAnalyzer", "_score_responsibility_clarity", context,
8599:             entities=consolidated_entities
8600:         )
8601:         # Semantic traceability via embeddings
8602:         semantic_chunks = self._execute_method(
8603:             "PolicyAnalysisEmbedder", "process_document", context,
8604:             document_text=document_text,
8605:             document_metadata=document_metadata
8606:         )
8607:         pdq_query = self._execute_method(
8608:             "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
8609:             pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
8610:         )
8611:         semantic_hits = self._execute_method(
8612:             "PolicyAnalysisEmbedder", "semantic_search", context,
8613:             query=pdq_query,
8614:             document_chunks=semantic_chunks or []
8615:         )
8616:         diversified_hits = self._execute_method(
8617:             "PolicyAnalysisEmbedder", "_apply_mmr", context,
8618:             ranked_results=semantic_hits or []
8619:         )
```

```
8620:     semantic_cube_stub = self._execute_method(
8621:         "SemanticAnalyzer", "_empty_semantic_cube", context
8622:     )
8623:     domain_scores = self._execute_method(
8624:         "SemanticAnalyzer", "_classify_policy_domain", context,
8625:         segment=document_text
8626:     )
8627:     cross_cutting = self._execute_method(
8628:         "SemanticAnalyzer", "_classify_cross_cutting_themes", context,
8629:         segment=document_text
8630:     )
8631:     value_chain = self._execute_method(
8632:         "SemanticAnalyzer", "_classify_value_chain_link", context,
8633:         segment=document_text
8634:     )
8635:     semantic_vectors = self._execute_method(
8636:         "SemanticAnalyzer", "_vectorize_segments", context,
8637:         segments=[document_text]
8638:     )
8639:     processed_segment = self._execute_method(
8640:         "SemanticAnalyzer", "_process_segment", context,
8641:         segment=document_text,
8642:         idx=0,
8643:         vector=semantic_vectors[0] if semantic_vectors else None
8644:     )
8645:     semantic_complexity = self._execute_method(
8646:         "SemanticAnalyzer", "_calculate_semantic_complexity", context,
8647:         semantic_cube=semantic_cube_stub
8648:     )
8649:     evidence_confidence = self._execute_method(
8650:         "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
8651:         matches=[m.get("bpin", "") for m in program_matches if isinstance(m, dict)],
8652:         text_length=len(document_text),
8653:         pattern_specificity=0.5
8654:     )
8655:     entity_dicts = [
8656:         self._execute_method("PDET Municipal Plan Analyzer", "_entity_to_dict", context, entity=e)
8657:         for e in consolidated_entities[:5]
8658:         if isinstance(e, dict) or hasattr(e, "__dict__")
8659:     ]
8660:
8661:     # Step 3: Generate traceability records
8662:     traceability_record = self._execute_method(
8663:         "AdaptivePriorCalculator", "generate_traceability_record", context,
8664:         matches=program_matches
8665:     )
8666:
8667:     # Step 4: Generate PDQ report
8668:     pdq_report = self._execute_method(
8669:         "PolicyAnalysisEmbedder", "generate_pdq_report", context,
8670:         traceability=traceability_record
8671:     )
8672:
8673:     # Step 5: Generate accountability matrix
8674:     accountability_matrix = self._execute_method(
8675:         "ReportingEngine", "generate_accountability_matrix", context,
```

```

8676:         entities=consolidated_entities
8677:     )
8678:
8679:     raw_evidence = {
8680:         "budgetary_traceability": {
8681:             "bpin_codes": [m.get("bpin") for m in program_matches if m.get("bpin")],
8682:             "project_codes": [m.get("project_code") for m in program_matches if m.get("project_code")],
8683:             "budget_matches": goal_budget_matches
8684:         },
8685:         "organizational_traceability": {
8686:             "responsible_entities": consolidated_entities,
8687:             "office_assignments": [e for e in consolidated_entities if e.get("office")],
8688:             "secretariat_assignments": [e for e in consolidated_entities if e.get("secretariat")]
8689:         },
8690:         "traceability_record": traceability_record,
8691:         "pdq_report": pdq_report,
8692:         "accountability_matrix": accountability_matrix,
8693:         "responsible_entities": responsible_entities,
8694:         "responsibility_clarity_score": responsibility_clarity,
8695:         "semantic_traceability": {
8696:             "query": pdq_query,
8697:             "semantic_hits": semantic_hits,
8698:             "diversified_hits": diversified_hits
8699:         },
8700:         "semantic_cube_baseline": semantic_cube_stub,
8701:         "policy_domain_scores": domain_scores,
8702:         "responsibility_entities_dict": entity_dicts,
8703:         "cross_cutting_themes": cross_cutting,
8704:         "value_chain_links": value_chain,
8705:         "semantic_vectors": semantic_vectors,
8706:         "semantic_complexity": semantic_complexity,
8707:         "evidence_confidence": evidence_confidence,
8708:         "processed_segment": processed_segment
8709:     }
8710:
8711:     return {
8712:         "executor_id": self.executor_id,
8713:         "raw_evidence": raw_evidence,
8714:         "metadata": {
8715:             "methods_executed": [log["method"] for log in self.execution_log],
8716:             "products_with_bpin": len([m for m in program_matches if m.get("bpin")]),
8717:             "products_with_responsible": len(consolidated_entities),
8718:             "canonical_question": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
8719:             "dimension_code": dim_info.code,
8720:             "dimension_label": dim_info.label
8721:         },
8722:         "execution_metrics": {
8723:             "methods_count": len(self.execution_log),
8724:             "all_succeeded": all(log["success"] for log in self.execution_log)
8725:         }
8726:     }
8727:
8728:
8729: class D3_Q4_TechnicalFeasibilityEvaluator(BaseExecutor):
8730:     """
8731:         DIM03_Q04_TECHNICAL_FEASIBILITY â\200\224 Evaluates activity-product feasibility vs resources/deadlines (canonical D3).

```

```
8732:     Epistemic mix: structural DAG validity, causal necessity, performance/implementation readiness, and statistical robustness.
8733:
8734:     Methods (from D3-Q4):
8735:         - AdvancedDAGValidator._calculate_statistical_power
8736:         - AdvancedDAGValidator._initialize_rng
8737:         - AdvancedDAGValidator.export_nodes
8738:         - AdvancedDAGValidator._generate_subgraph
8739:         - AdvancedDAGValidator._get_node_validator
8740:         - AdvancedDAGValidator._create_empty_result
8741:         - HierarchicalGenerativeModel._calculate_likelihood
8742:         - IndustrialGradeValidator.validate_causal_categories
8743:         - TeoriaCambio._extraer_categorias
8744:         - AdvancedDAGValidator.get_graph_stats
8745:         - AdvancedDAGValidator._calculate_node_importance
8746:         - AdvancedDAGValidator.calculate_acyclicity_pvalue
8747:         - AdvancedDAGValidator._is_acyclic
8748:         - BayesianMechanismInference._test_necessity
8749:         - IndustrialGradeValidator.execute_suite
8750:         - IndustrialGradeValidator.validate_connection_matrix
8751:         - IndustrialGradeValidator.run_performance_benchmarks
8752:         - IndustrialGradeValidator._benchmark_operation
8753:         - PerformanceAnalyzer.analyze_performance
8754:         - PerformanceAnalyzer._calculate_loss_functions
8755:         - HierarchicalGenerativeModel._calculate_ess
8756:
8757:
8758:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8759:         raw_evidence = {}
8760:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
8761:         plan_name = context.get("metadata", {}).get("title", "plan_desarrollo")
8762:
8763:         # Step 1: Validate DAG structure
8764:         acyclicity_pvalue = self._execute_method(
8765:             "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context
8766:         )
8767:         is_acyclic = self._execute_method(
8768:             "AdvancedDAGValidator", "_is_acyclic", context
8769:         )
8770:         graph_stats = self._execute_method(
8771:             "AdvancedDAGValidator", "get_graph_stats", context
8772:         )
8773:         node_importance = self._execute_method(
8774:             "AdvancedDAGValidator", "_calculate_node_importance", context
8775:         )
8776:         subgraph = self._execute_method(
8777:             "AdvancedDAGValidator", "_generate_subgraph", context
8778:         )
8779:         added_node = self._execute_method(
8780:             "AdvancedDAGValidator", "add_node", context,
8781:             node_name="temp_node"
8782:         )
8783:         added_edge = self._execute_method(
8784:             "AdvancedDAGValidator", "add_edge", context,
8785:             source="temp_node",
8786:             target="temp_target",
8787:             weight=1.0
```

```
8788:         )
8789:         node_export = self._execute_method(
8790:             "AdvancedDAGValidator", "export_nodes", context
8791:         )
8792:         rng_seed = self._execute_method(
8793:             "AdvancedDAGValidator", "_initialize_rng", context,
8794:             plan_name=plan_name,
8795:             salt=dim_info.code
8796:         )
8797:         stat_power = self._execute_method(
8798:             "AdvancedDAGValidator", "_calculate_statistical_power", context,
8799:             s=int(graph_stats.get("edges", 0)),
8800:             n=max(1, int(graph_stats.get("nodes", 1)))
8801:         )
8802:         node_validator = self._execute_method(
8803:             "AdvancedDAGValidator", "get_node_validator", context,
8804:             node_type="producto"
8805:         )
8806:         empty_result = self._execute_method(
8807:             "AdvancedDAGValidator", "_create_empty_result", context,
8808:             plan_name=plan_name,
8809:             seed=rng_seed,
8810:             timestamp=context.get("metadata", {}).get("timestamp", ""))
8811:     )
8812:
8813:     # Step 2: Test necessity of activities for products
8814:     necessity_test = self._execute_method(
8815:         "BayesianMechanismInference", "_test_necessity", context
8816:     )
8817:
8818:     # Step 3: Execute industrial-grade validation
8819:     validation_suite = self._execute_method(
8820:         "IndustrialGradeValidator", "execute_suite", context
8821:     )
8822:     connection_validation = self._execute_method(
8823:         "IndustrialGradeValidator", "validate_connection_matrix", context
8824:     )
8825:     performance_benchmarks = self._execute_method(
8826:         "IndustrialGradeValidator", "run_performance_benchmarks", context
8827:     )
8828:     benchmark_ops = self._execute_method(
8829:         "IndustrialGradeValidator", "_benchmark_operation", context
8830:     )
8831:     metric_log = self._execute_method(
8832:         "IndustrialGradeValidator", "_log_metric", context,
8833:         name="custom_latency",
8834:         value=graph_stats.get("edges", 0),
8835:         unit="edges",
8836:         threshold=10.0
8837:     )
8838:     engine_readiness = self._execute_method(
8839:         "IndustrialGradeValidator", "validate_engine_readiness", context
8840:     )
8841:
8842:     # Step 4: Analyze performance
8843:     performance_analysis = self._execute_method(
```

```
8844:         "PerformanceAnalyzer", "analyze_performance", context
8845:     )
8846:     loss_functions = self._execute_method(
8847:         "PerformanceAnalyzer", "_calculate_loss_functions", context
8848:     )
8849:     # Likelihood estimation for resource adequacy
8850:     resource_likelihood = self._execute_method(
8851:         "HierarchicalGenerativeModel", "_calculate_likelihood", context,
8852:         mechanism_type="tecnico",
8853:         observations={"coherence": performance_analysis.get("resource_fit", {}).get("score", 0.0)}
8854:     )
8855:
8856:     # Step 5: Calculate effective sample size
8857:     ess = self._execute_method(
8858:         "HierarchicalGenerativeModel", "_calculate_ess", context
8859:     )
8860:     r_hat = self._execute_method(
8861:         "HierarchicalGenerativeModel", "_calculate_r_hat", context,
8862:         chains=[]
8863:     )
8864:     causal_categories_valid = self._execute_method(
8865:         "IndustrialGradeValidator", "validate_causal_categories", context
8866:     )
8867:     extracted_categories = self._execute_method(
8868:         "TeoriaCambio", "_extraer_categorias", context,
8869:         text=context.get("document_text", "")
8870:     )
8871:
8872:     raw_evidence = {
8873:         "activity_product_mapping": connection_validation,
8874:         "resource_adequacy": performance_analysis.get("resource_fit", {}),
8875:         "timeline_feasibility": performance_analysis.get("timeline_feasibility", {}),
8876:         "technical_validation": {
8877:             "dag_valid": is_acyclic,
8878:             "acyclicity_p": acyclicity_pvalue,
8879:             "necessity_score": necessity_test,
8880:             "graph_stats": graph_stats,
8881:             "node_importance": node_importance,
8882:             "subgraph_sample": subgraph,
8883:             "added_node": added_node,
8884:             "added_edge": added_edge,
8885:             "node_validator": node_validator,
8886:             "empty_result": empty_result,
8887:             "node_export": node_export,
8888:             "rng_seed": rng_seed,
8889:             "statistical_power": stat_power
8890:         },
8891:         "performance_metrics": {
8892:             "benchmarks": performance_benchmarks,
8893:             "loss_functions": loss_functions,
8894:             "ess": ess,
8895:             "r_hat": r_hat,
8896:             "resource_likelihood": resource_likelihood
8897:         },
8898:         "engine_readiness": engine_readiness,
8899:         "feasibility_score": validation_suite.get("overall_score", 0),
```

```
8900:         "causal_categories_valid": causal_categories_valid,
8901:         "extracted_categories": extracted_categories,
8902:         "metric_log": metric_log
8903:     }
8904:
8905:     return {
8906:         "executor_id": self.executor_id,
8907:         "raw_evidence": raw_evidence,
8908:         "metadata": {
8909:             "methods_executed": [log["method"] for log in self.execution_log],
8910:             "dag_is_valid": is_acyclic,
8911:             "feasibility_score": validation_suite.get("overall_score", 0),
8912:             "canonical_question": "DIM03_Q04_TECHNICAL_FEASIBILITY",
8913:             "dimension_code": dim_info.code,
8914:             "dimension_label": dim_info.label
8915:         },
8916:         "execution_metrics": {
8917:             "methods_count": len(self.execution_log),
8918:             "all_succeeded": all(log["success"] for log in self.execution_log)
8919:         }
8920:     }
8921:
8922:
8923: class D3_Q5_OutputOutcomeLinkageAnalyzer(BaseExecutor):
8924:     """
8925:         DIM03_Q05_OUTPUT_OUTCOME_LINKAGE \200\224 Analyzes mechanisms linking outputs to outcomes with canonical D3 labeling.
8926:         Epistemic mix: semantic hierarchy checks, causal order validation, DAG/effect estimation, and Bayesian mechanism inference.
8927:
8928:     Methods (from D3-Q5):
8929:         - PDET MunicipalPlanAnalyzer._identify_confounders
8930:         - PDET MunicipalPlanAnalyzer._effect_to_dict
8931:         - PDET MunicipalPlanAnalyzer._scenario_to_dict
8932:         - PDET MunicipalPlanAnalyzer._simulate_intervention
8933:         - PDET MunicipalPlanAnalyzer._generate_recommendations
8934:         - PDET MunicipalPlanAnalyzer._identify_causal_nodes
8935:         - BayesianCounterfactualAuditor._evaluate_factual
8936:         - BayesianCounterfactualAuditor._evaluate_counterfactual
8937:         - CausalExtractor._assess_financial_consistency
8938:         - BayesianMechanismInference._infer_activity_sequence
8939:         - BayesianMechanismInference._generate_necessity_remediation
8940:         - BayesianCounterfactualAuditor.refutation_and_sanity_checks
8941:         - IndustrialPolicyProcessor._load_questionnaire
8942:         - PDET MunicipalPlanAnalyzer.analyze_financial_feasibility
8943:         - PDET MunicipalPlanAnalyzer.construct_causal_dag
8944:         - PDET MunicipalPlanAnalyzer.estimate_causal_effects
8945:         - PDET MunicipalPlanAnalyzer.generate_counterfactuals
8946:         - CausalExtractor._build_type_hierarchy
8947:         - CausalExtractor._check_structuralViolation
8948:         - CausalExtractor._calculate_type_transition_prior
8949:         - CausalExtractor._calculate_textual_proximity
8950:         - TeoriaCambio._validar orden_causal
8951:         - PDET MunicipalPlanAnalyzer._refine_edge_probabilities
8952:         - PolicyAnalysisEmbedder.compare_policy_interventions
8953:         - BayesianMechanismInference.infer_mechanisms
8954:     """
8955:
```

```
8956:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
8957:         raw_evidence = {}
8958:         dim_info = get_dimension_info(CanonicalDimension.D3.value)
8959:
8960:         # Step 0: Build causal backbone and effects
8961:         financial_analysis = self._execute_method(
8962:             "PDETMunicipalPlanAnalyzer", "analyze_financial_feasibility", context
8963:         )
8964:         causal_dag = self._execute_method(
8965:             "PDETMunicipalPlanAnalyzer", "construct_causal_dag", context,
8966:             financial_analysis=financial_analysis
8967:         )
8968:         causal_effects = self._execute_method(
8969:             "PDETMunicipalPlanAnalyzer", "estimate_causal_effects", context,
8970:             dag=causal_dag,
8971:             financial_analysis=financial_analysis
8972:         )
8973:         counterfactuals = self._execute_method(
8974:             "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context,
8975:             dag=causal_dag,
8976:             causal_effects=causal_effects,
8977:             financial_analysis=financial_analysis
8978:         )
8979:         simulated_intervention = self._execute_method(
8980:             "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,
8981:             intervention={},
8982:             dag=causal_dag,
8983:             causal_effects=causal_effects,
8984:             label="baseline"
8985:         )
8986:         causal_nodes = self._execute_method(
8987:             "PDETMunicipalPlanAnalyzer", "_identify_causal_nodes", context,
8988:             text=context.get("document_text", ""),
8989:             tables=context.get("tables", []),
8990:             financial_analysis=financial_analysis
8991:         )
8992:         confounders = {}
8993:         for effect in causal_effects:
8994:             treatment = effect.treatment if hasattr(effect, "treatment") else None
8995:             outcome = effect.outcome if hasattr(effect, "outcome") else None
8996:             if treatment and outcome:
8997:                 confounders[(treatment, outcome)] = self._execute_method(
8998:                     "PDETMunicipalPlanAnalyzer", "_identify_confounders", context,
8999:                     treatment=treatment,
9000:                     outcome=outcome,
9001:                     dag=causal_dag
9002:                 )
9003:             effect_dicts = [
9004:                 self._execute_method("PDETMunicipalPlanAnalyzer", "_effect_to_dict", context, effect=effect)
9005:                 for effect in causal_effects
9006:             ]
9007:             scenario_dicts = [
9008:                 self._execute_method("PDETMunicipalPlanAnalyzer", "_scenario_to_dict", context, scenario=scenario)
9009:                 for scenario in counterfactuals
9010:             ]
9011:             causal_recommendations = self._execute_method(
```

```

9012:         "PDETMunicipalPlanAnalyzer", "_generate_recommendations", context,
9013:         analysis_results={"financial_analysis": financial_analysis, "quality_score": getattr(causal_dag, 'graph', {})}
9014:     )
9015:     financial_consistency = None
9016:     if refined_edges:
9017:         first_edge = refined_edges[0] if isinstance(refined_edges, list) else {}
9018:         source = first_edge.get("source") if isinstance(first_edge, dict) else ""
9019:         target = first_edge.get("target") if isinstance(first_edge, dict) else ""
9020:         financial_consistency = self._execute_method(
9021:             "CausalExtractor", "_assess_financial_consistency", context,
9022:             source=source or "",
9023:             target=target or ""
9024:         )
9025:     factual_eval = None
9026:     counterfactual_eval = None
9027:     if causal_effects:
9028:         first_effect = causal_effects[0]
9029:         target = getattr(first_effect, "outcome", None) or ""
9030:         evidence = {"p_effect": getattr(first_effect, "probability_significant", 0.0)}
9031:         factual_eval = self._execute_method(
9032:             "BayesianCounterfactualAuditor", "_evaluate_factual", context,
9033:             target=target,
9034:             evidence=evidence
9035:         )
9036:         counterfactual_eval = self._execute_method(
9037:             "BayesianCounterfactualAuditor", "_evaluate_counterfactual", context,
9038:             target=target,
9039:             intervention={"shift": 0.1}
9040:         )
9041:     matched_node = None
9042:     try:
9043:         matched_node = self._execute_method(
9044:             "PDETMunicipalPlanAnalyzer", "_match_text_to_node", context,
9045:             text=context.get("document_text", "")[:200],
9046:             nodes=causal_nodes if isinstance(causal_nodes, dict) else {}
9047:         )
9048:     except Exception:
9049:         matched_node = None
9050:
9051:     # Step 1: Build type hierarchy
9052:     type_hierarchy = self._execute_method(
9053:         "CausalExtractor", "_build_type_hierarchy", context
9054:     )
9055:
9056:     # Step 2: Check structural violations
9057:     structural_violations = self._execute_method(
9058:         "CausalExtractor", "_check_structuralViolation", context,
9059:         hierarchy=type_hierarchy
9060:     )
9061:
9062:     # Step 3: Calculate transition priors and proximity
9063:     transition_priors = self._execute_method(
9064:         "CausalExtractor", "_calculate_type_transition_prior", context,
9065:         hierarchy=type_hierarchy
9066:     )
9067:     textual_proximity = self._execute_method(

```

```
9068:         "CausalExtractor", "_calculate_textual_proximity", context
9069:     )
9070:
9071:     # Step 4: Validate causal order
9072:     causal_order_validation = self._execute_method(
9073:         "TeoriaCambio", "_validar_orden_causal", context,
9074:         hierarchy=type_hierarchy
9075:     )
9076:
9077:     # Step 5: Refine edge probabilities
9078:     refined_edges = self._execute_method(
9079:         "PDET MunicipalPlanAnalyzer", "_refine_edge_probabilities", context,
9080:         priors=transition_priors
9081:     )
9082:
9083:     # Step 6: Compare policy interventions
9084:     intervention_comparison = self._execute_method(
9085:         "PolicyAnalysisEmbedder", "compare_policy_interventions", context
9086:     )
9087:
9088:     # Step 7: Infer mechanisms
9089:     mechanisms = self._execute_method(
9090:         "BayesianMechanismInference", "infer_mechanisms", context,
9091:         edges=refined_edges
9092:     )
9093:     mechanism_sample = next(iter(mechanisms.values()), {})
9094:     activity_sequence = self._execute_method(
9095:         "BayesianMechanismInference", "_infer_activity_sequence", context,
9096:         observations=mechanism_sample.get("observations", {}),
9097:         mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0})
9098:     )
9099:     quantified_uncertainty = self._execute_method(
9100:         "BayesianMechanismInference", "_quantify_uncertainty", context,
9101:         mechanism_type_posterior=mechanism_sample.get("mechanism_type", {"tecnico": 1.0}),
9102:         sequence_posterior=mechanism_sample.get("activity_sequence", {}),
9103:         coherence_score=mechanism_sample.get("coherence_score", 0.0)
9104:     )
9105:     mechanism_observations = self._execute_method(
9106:         "BayesianMechanismInference", "_extract_observations", context,
9107:         node={"id": next(iter(mechanisms.keys()), "")},
9108:         text=context.get("document_text", "")
9109:     )
9110:     necessity_remediation = self._execute_method(
9111:         "BayesianMechanismInference", "_generate_necessity_remediation", context,
9112:         node_id=next(iter(mechanisms.keys()), ""),
9113:         missing_components=structural_violations
9114:     )
9115:     questionnaire_stub = self._execute_method(
9116:         "IndustrialPolicyProcessor", "_load_questionnaire", context
9117:     )
9118:     reputation_checks = None
9119:     try:
9120:         confounder_keys = list(confounders.keys())
9121:         first_pair = confounder_keys[0] if confounder_keys else ("", "")
9122:         reputation_checks = self._execute_method(
9123:             "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
```

```
9124:         dag=getattr(causal_dag, "graph", None),
9125:         target=first_pair[1],
9126:         treatment=first_pair[0],
9127:         confounders=list(confounders.values())[0] if confounders else []
9128:     )
9129: except Exception:
9130:     refutation_checks = None
9131:
9132: raw_evidence = {
9133:     "output_outcome_links": refined_edges,
9134:     "mechanism_explanation": mechanisms,
9135:     "type_hierarchy": type_hierarchy,
9136:     "causal_dag": causal_dag,
9137:     "causal_effects": causal_effects,
9138:     "counterfactuals": counterfactuals,
9139:     "simulated_intervention": simulated_intervention,
9140:     "causal_nodes": causal_nodes,
9141:     "financial_analysis": financial_analysis,
9142:     "causal_validity": {
9143:         "structural_violations": structural_violations,
9144:         "order_valid": causal_order_validation
9145:     },
9146:     "transition_probabilities": transition_priors,
9147:     "textual_proximity": textual_proximity,
9148:     "intervention_comparison": intervention_comparison,
9149:     "confounders": confounders,
9150:     "effect_dicts": effect_dicts,
9151:     "scenario_dicts": scenario_dicts,
9152:     "activity_sequence_sample": activity_sequence,
9153:     "uncertainty_quantified": quantified_uncertainty,
9154:     "mechanism_observations": mechanism_observations,
9155:     "refutation_checks": refutation_checks,
9156:     "necessity_remediation": necessity_remediation,
9157:     "questionnaire_stub": questionnaire_stub,
9158:     "causal_recommendations": causal_recommendations,
9159:     "financial_consistency": financial_consistency,
9160:     "factual_eval": factual_eval,
9161:     "counterfactual_eval": counterfactual_eval,
9162:     "matched_node": matched_node
9163: }
9164:
9165: return {
9166:     "executor_id": self.executor_id,
9167:     "raw_evidence": raw_evidence,
9168:     "metadata": {
9169:         "methods_executed": [log["method"] for log in self.execution_log],
9170:         "mechanisms_identified": len(mechanisms),
9171:         "violations_found": len(structuralViolations),
9172:         "canonical_question": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
9173:         "dimension_code": dim_info.code,
9174:         "dimension_label": dim_info.label
9175:     },
9176:     "execution_metrics": {
9177:         "methods_count": len(self.execution_log),
9178:         "all_succeeded": all(log["success"] for log in self.execution_log)
9179:     }
}
```

```
9180:         }
9181:
9182:
9183: # =====
9184: # DIMENSION 4: RESULTS & OUTCOMES
9185: # =====
9186:
9187: class D4_Q1_OutcomeMetricsValidator(BaseExecutor):
9188:     """
9189:         DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS \200\224 Validates outcome indicators (baseline, target, horizon) with canonical D4 notation.
9190:         Epistemic mix: semantic goal extraction, temporal/consistency checks, statistical performance signals, and indicator quality scoring.
9191:
9192:     Methods (from D4-Q1):
9193:         - PDETMunicipalPlanAnalyzer._extract_entities_syntax
9194:         - PDETMunicipalPlanAnalyzer._extract_entities_ner
9195:         - CausalExtractor._calculate_language_specificity
9196:         - CausalExtractor._calculate_composite_likelihood
9197:         - CausalExtractor._calculate_semantic_distance
9198:         - TemporalLogicVerifier._classify_temporal_type
9199:         - PDETMunicipalPlanAnalyzer._score_indicators
9200:         - PDETMunicipalPlanAnalyzer._find_outcome_mentions
9201:         - PDETMunicipalPlanAnalyzer._score_temporal_consistency
9202:         - CausalExtractor._extract_goals
9203:         - CausalExtractor._parse_goal_context
9204:         - CausalExtractor._classify_goal_type
9205:         - TemporalLogicVerifier._parse_temporal_marker
9206:         - TemporalLogicVerifier._extract_resources
9207:         - PerformanceAnalyzer.analyze_performance
9208:         - PerformanceAnalyzer._generate_recommendations
9209:     """
9210:
9211:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9212:         raw_evidence = {}
9213:         dim_info = get_dimension_info(CanonicalDimension.D4.value)
9214:
9215:         # Step 1: Find outcome mentions
9216:         outcome_mentions = self._execute_method(
9217:             "PDETMunicipalPlanAnalyzer", "_find_outcome_mentions", context
9218:         )
9219:         entities_syntax = self._execute_method(
9220:             "PDETMunicipalPlanAnalyzer", "_extract_entities_syntax", context,
9221:             text=context.get("document_text", ""))
9222:         )
9223:         entities_ner = self._execute_method(
9224:             "PDETMunicipalPlanAnalyzer", "_extract_entities_ner", context,
9225:             text=context.get("document_text", ""))
9226:         )
9227:
9228:         # Step 2: Score temporal consistency
9229:         temporal_consistency = self._execute_method(
9230:             "PDETMunicipalPlanAnalyzer", "_score_temporal_consistency", context,
9231:             outcomes=outcome_mentions
9232:         )
9233:
9234:         # Step 3: Extract and classify goals
9235:         goals = self._execute_method(
```

```
9236:         "CausalExtractor", "_extract_goals", context
9237:     )
9238:     goal_contexts = self._execute_method(
9239:         "CausalExtractor", "_parse_goal_context", context,
9240:         goals=goals
9241:     )
9242:     goal_types = self._execute_method(
9243:         "CausalExtractor", "_classify_goal_type", context,
9244:         goals=goals
9245:     )
9246:     semantic_distance = 0.0
9247:     if goal_types and outcome_mentions:
9248:         semantic_distance = self._execute_method(
9249:             "CausalExtractor", "_calculate_semantic_distance", context,
9250:             source=str(goal_types[0]),
9251:             target=str(outcome_mentions[0])
9252:         )
9253:
9254:     # Step 4: Parse temporal markers
9255:     temporal_markers = self._execute_method(
9256:         "TemporalLogicVerifier", "_parse_temporal_marker", context,
9257:         contexts=goal_contexts
9258:     )
9259:     temporal_type = self._execute_method(
9260:         "TemporalLogicVerifier", "_classify_temporal_type", context,
9261:         marker=temporal_markers[0] if temporal_markers else ""
9262:     )
9263:     resources_mentioned = self._execute_method(
9264:         "TemporalLogicVerifier", "_extract_resources", context,
9265:         text=context.get("document_text", "")
9266:     )
9267:     precedence_check = self._execute_method(
9268:         "TemporalLogicVerifier", "_should_precede", context,
9269:         marker_a=temporal_markers[0] if temporal_markers else "",
9270:         marker_b=temporal_markers[1] if len(temporal_markers) > 1 else ""
9271:     )
9272:
9273:     # Step 5: Analyze performance
9274:     performance_analysis = self._execute_method(
9275:         "PerformanceAnalyzer", "analyze_performance", context,
9276:         outcomes=outcome_mentions
9277:     )
9278:     indicator_quality = self._execute_method(
9279:         "PDETMunicipalPlanAnalyzer", "_score_indicators", context
9280:     )
9281:     performance_recommendations = self._execute_method(
9282:         "PerformanceAnalyzer", "generate_recommendations", context,
9283:         performance_analysis=performance_analysis
9284:     )
9285:     # Semantic certainty for goals
9286:     language_specificity = self._execute_method(
9287:         "CausalExtractor", "_calculate_language_specificity", context,
9288:         keyword=goal_contexts[0] if goal_contexts else "",
9289:         policy_area=context.get("policy_area")
9290:     )
9291:     composite_likelihood = self._execute_method(
```

```

9292:         "CausalExtractor", "_calculate_composite_likelihood", context,
9293:         evidence={
9294:             "semantic_distance": indicator_quality if isinstance(indicator_quality, (int, float)) else 0.0,
9295:             "textual_proximity": performance_analysis.get("coherence_score", 0.0) if isinstance(performance_analysis, dict) else 0.0,
9296:             "language_specificity": language_specificity,
9297:             "temporal_coherence": temporal_consistency if isinstance(temporal_consistency, (int, float)) else 0.0
9298:         }
9299:     )
9300:
9301:     raw_evidence = {
9302:         "outcome_indicators": outcome_mentions,
9303:         "indicators_with_baseline": [o for o in outcome_mentions if o.get("has_baseline")],
9304:         "indicators_with_target": [o for o in outcome_mentions if o.get("has_target")],
9305:         "indicators_with_horizon": [o for o in outcome_mentions if o.get("time_horizon")],
9306:         "temporal_consistency_score": temporal_consistency,
9307:         "goal_classifications": goal_types,
9308:         "temporal_markers": temporal_markers,
9309:         "performance_metrics": performance_analysis,
9310:         "indicator_quality": indicator_quality,
9311:         "performance_recommendations": performance_recommendations,
9312:         "entities_syntax": entities_syntax,
9313:         "entities_ner": entities_ner,
9314:         "temporal_type": temporal_type,
9315:         "language_specificity": language_specificity,
9316:         "composite_likelihood": composite_likelihood,
9317:         "goal_outcome_semantic_distance": semantic_distance,
9318:         "resources_mentioned": resources_mentioned,
9319:         "precedence_check": precedence_check
9320:     }
9321:
9322:     return {
9323:         "executor_id": self.executor_id,
9324:         "raw_evidence": raw_evidence,
9325:         "metadata": {
9326:             "methods_executed": [log["method"] for log in self.execution_log],
9327:             "total_outcomes": len(outcome_mentions),
9328:             "complete_indicators": len([o for o in outcome_mentions
9329:                 if o.get("has_baseline") and o.get("has_target") and o.get("time_horizon")]),
9330:             "canonical_question": "DIM04_Q01_OUTCOME_INDICATOR完整性",
9331:             "dimension_code": dim_info.code,
9332:             "dimension_label": dim_info.label
9333:         },
9334:         "execution_metrics": {
9335:             "methods_count": len(self.execution_log),
9336:             "all_succeeded": all(log["success"] for log in self.execution_log)
9337:         }
9338:     }
9339:
9340:
9341: class D4_Q2_CausalChainValidator(BaseExecutor):
9342:     """
9343:     Validates explicit causal chain with assumptions and enabling conditions.
9344:
9345:     Methods (from D4-Q2):
9346:     - TeoriaCambio._encontrar_caminos_completos
9347:     - TeoriaCambio.validacion_completa

```

```
9348:     - CausalExtractor.extract_causal_hierarchy
9349:     - HierarchicalGenerativeModel.verify_conditional_independence
9350:     - HierarchicalGenerativeModel._generate_independence_tests
9351:     - BayesianCounterfactualAuditor.construct_scm
9352:     - AdvancedDAGValidator._perform_sensitivity_analysis_internal
9353:     - BayesFactorTable.get_bayes_factor
9354:
9355: """
9356: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9357:     raw_evidence = {}
9358:
9359:     # Step 1: Find complete causal paths
9360:     complete_paths = self._execute_method(
9361:         "TeoriaCambio", "_encontrar_caminos_completos", context
9362:     )
9363:
9364:     # Step 2: Complete validation
9365:     validation_results = self._execute_method(
9366:         "TeoriaCambio", "validacion_completa", context,
9367:         paths=complete_paths
9368:     )
9369:
9370:     # Step 3: Extract causal hierarchy
9371:     causal_hierarchy = self._execute_method(
9372:         "CausalExtractor", "extract_causal_hierarchy", context
9373:     )
9374:
9375:     # Step 4: Verify conditional independence
9376:     independence_verification = self._execute_method(
9377:         "HierarchicalGenerativeModel", "verify_conditional_independence", context,
9378:         hierarchy=causal_hierarchy
9379:     )
9380:     independence_tests = self._execute_method(
9381:         "HierarchicalGenerativeModel", "_generate_independence_tests", context,
9382:         verification=independence_verification
9383:     )
9384:
9385:     # Step 5: Construct structural causal model
9386:     scm = self._execute_method(
9387:         "BayesianCounterfactualAuditor", "construct_scm", context,
9388:         hierarchy=causal_hierarchy
9389:     )
9390:
9391:     # Step 6: Perform sensitivity analysis
9392:     sensitivity_analysis = self._execute_method(
9393:         "AdvancedDAGValidator", "_perform_sensitivity_analysis_internal", context,
9394:         scm=scm
9395:     )
9396:
9397:     # Step 7: Get Bayes factor
9398:     bayes_factor = self._execute_method(
9399:         "BayesFactorTable", "get_bayes_factor", context,
9400:         analysis=sensitivity_analysis
9401:     )
9402:
9403:     raw_evidence = {
```

```

9404:         "causal_chain": complete_paths,
9405:         "key_assumptions": validation_results.get("assumptions", []),
9406:         "enabling_conditions": validation_results.get("conditions", []),
9407:         "external_factors": validation_results.get("external_factors", []),
9408:         "causal_hierarchy": causal_hierarchy,
9409:         "independence_tests": independence_tests,
9410:         "structural_model": scm,
9411:         "sensitivity": sensitivity_analysis,
9412:         "evidential_strength": bayes_factor
9413:     }
9414:
9415:     return {
9416:         "executor_id": self.executor_id,
9417:         "raw_evidence": raw_evidence,
9418:         "metadata": {
9419:             "methods_executed": [log["method"] for log in self.execution_log],
9420:             "complete_paths_found": len(complete_paths),
9421:             "assumptions_identified": len(validation_results.get("assumptions", []))
9422:         },
9423:         "execution_metrics": {
9424:             "methods_count": len(self.execution_log),
9425:             "all_succeeded": all(log["success"] for log in self.execution_log)
9426:         }
9427:     }
9428:
9429:
9430: class D4_Q3_AmbitionJustificationAnalyzer(BaseExecutor):
9431:     """
9432:     Analyzes justification of result ambition based on investment/capacity/benchmarks.
9433:
9434:     Methods (from D4-Q3):
9435:     - PDETMunicipalPlanAnalyzer._get_prior_effect
9436:     - PDETMunicipalPlanAnalyzer._estimate_effect_bayesian
9437:     - PDETMunicipalPlanAnalyzer._compute_robustness_value
9438:     - AdaptivePriorCalculator.sensitivity_analysis
9439:     - HierarchicalGenerativeModel._calculate_r_hat
9440:     - HierarchicalGenerativeModel._calculate_ess
9441:     - AdvancedDAGValidator._calculate_statistical_power
9442:     - BayesianMechanismInference._aggregate_bayesian_confidence
9443:     """
9444:
9445:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9446:         raw_evidence = {}
9447:
9448:         # Step 1: Get prior effect estimates
9449:         prior_effects = self._execute_method(
9450:             "PDETMunicipalPlanAnalyzer", "_get_prior_effect", context
9451:         )
9452:
9453:         # Step 2: Estimate effect using Bayesian methods
9454:         effect_estimate = self._execute_method(
9455:             "PDETMunicipalPlanAnalyzer", "_estimate_effect_bayesian", context,
9456:             priors=prior_effects
9457:         )
9458:
9459:         # Step 3: Compute robustness

```

```
9460:     robustness = self._execute_method(
9461:         "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
9462:         estimate=effect_estimate
9463:     )
9464:
9465:     # Step 4: Sensitivity analysis
9466:     sensitivity = self._execute_method(
9467:         "AdaptivePriorCalculator", "sensitivity_analysis", context,
9468:         estimate=effect_estimate
9469:     )
9470:
9471:     # Step 5: Calculate convergence diagnostics
9472:     r_hat = self._execute_method(
9473:         "HierarchicalGenerativeModel", "_calculate_r_hat", context
9474:     )
9475:     ess = self._execute_method(
9476:         "HierarchicalGenerativeModel", "_calculate_ess", context
9477:     )
9478:
9479:     # Step 6: Calculate statistical power
9480:     statistical_power = self._execute_method(
9481:         "AdvancedDAGValidator", "_calculate_statistical_power", context,
9482:         effect=effect_estimate
9483:     )
9484:
9485:     # Step 7: Aggregate confidence
9486:     confidence_aggregate = self._execute_method(
9487:         "BayesianMechanismInference", "_aggregate_bayesian_confidence", context,
9488:         estimates=[effect_estimate, robustness, statistical_power]
9489:     )
9490:
9491:     raw_evidence = {
9492:         "ambition_level": context.get("target_ambition", {}),
9493:         "financial_investment": context.get("total_investment", 0),
9494:         "institutional_capacity": context.get("capacity_score", 0),
9495:         "comparative_benchmarks": prior_effects,
9496:         "justification_analysis": {
9497:             "effect_estimate": effect_estimate,
9498:             "robustness": robustness,
9499:             "sensitivity": sensitivity,
9500:             "statistical_power": statistical_power
9501:         },
9502:         "convergence_diagnostics": {
9503:             "r_hat": r_hat,
9504:             "ess": ess
9505:         },
9506:         "overall_confidence": confidence_aggregate
9507:     }
9508:
9509:     return {
9510:         "executor_id": self.executor_id,
9511:         "raw_evidence": raw_evidence,
9512:         "metadata": {
9513:             "methods_executed": [log["method"] for log in self.execution_log],
9514:             "ambition_justified": confidence_aggregate > 0.7,
9515:             "statistical_power": statistical_power
9516:         }
9517:     }
```

```
9516:         },
9517:         "execution_metrics": {
9518:             "methods_count": len(self.execution_log),
9519:             "all_succeeded": all(log["success"] for log in self.execution_log)
9520:         }
9521:     }
9522:
9523:
9524: class D4_Q4_ProblemSolvencyEvaluator(BaseExecutor):
9525:     """
9526:         Evaluates whether results address/resolve prioritized problems from diagnosis.
9527:
9528:         Methods (from D4-Q4):
9529:         - PolicyContradictionDetector._calculate_objective_alignment
9530:         - PolicyContradictionDetector._identify_affected_sections
9531:         - PolicyContradictionDetector._generate_resolution_recommendations
9532:         - OperationalizationAuditor._generate_optimal_remediations
9533:         - OperationalizationAuditor._get_remediation_text
9534:         - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
9535:         - FinancialAuditor._detect_allocation_gaps
9536:     """
9537:
9538:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9539:         raw_evidence = {}
9540:
9541:         # Step 1: Calculate objective alignment
9542:         objective_alignment = self._execute_method(
9543:             "PolicyContradictionDetector", "_calculate_objective_alignment", context
9544:         )
9545:
9546:         # Step 2: Identify affected sections
9547:         affected_sections = self._execute_method(
9548:             "PolicyContradictionDetector", "_identify_affected_sections", context,
9549:             alignment=objective_alignment
9550:         )
9551:
9552:         # Step 3: Generate resolution recommendations
9553:         resolutions = self._execute_method(
9554:             "PolicyContradictionDetector", "_generate_resolution_recommendations", context,
9555:             sections=affected_sections
9556:         )
9557:
9558:         # Step 4: Generate optimal remediations
9559:         remediations = self._execute_method(
9560:             "OperationalizationAuditor", "_generate_optimal_remediations", context
9561:         )
9562:         remediation_text = self._execute_method(
9563:             "OperationalizationAuditor", "_get_remediation_text", context,
9564:             remediations=remediations
9565:         )
9566:
9567:         # Step 5: Aggregate risk and prioritize
9568:         risk_priorities = self._execute_method(
9569:             "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context
9570:         )
9571:
```

```

9572:     # Step 6: Detect allocation gaps
9573:     allocation_gaps = self._execute_method(
9574:         "FinancialAuditor", "_detect_allocation_gaps", context
9575:     )
9576:
9577:     raw_evidence = {
9578:         "prioritized_problems": context.get("diagnosis_problems", []),
9579:         "proposed_results": context.get("outcome_indicators", []),
9580:         "problem_result_mapping": objective_alignment,
9581:         "unaddressed_problems": [p for p in affected_sections if not p.get("addressed")],
9582:         "solvency_score": objective_alignment.get("score", 0),
9583:         "resolution_recommendations": resolutions,
9584:         "remediations": remediation_text,
9585:         "risk_priorities": risk_priorities,
9586:         "allocation_gaps": allocation_gaps
9587:     }
9588:
9589:     return {
9590:         "executor_id": self.executor_id,
9591:         "raw_evidence": raw_evidence,
9592:         "metadata": {
9593:             "methods_executed": [log["method"] for log in self.execution_log],
9594:             "problems_addressed": len([p for p in affected_sections if p.get("addressed")]),
9595:             "problems_unaddressed": len([p for p in affected_sections if not p.get("addressed")])
9596:         },
9597:         "execution_metrics": {
9598:             "methods_count": len(self.execution_log),
9599:             "all_succeeded": all(log["success"] for log in self.execution_log)
9600:         }
9601:     }
9602:
9603:
9604: class D4_Q5_VerticalAlignmentValidator(BaseExecutor):
9605:     """
9606:     Validates alignment with superior frameworks (PND, SDGs).
9607:
9608:     Methods (from D4-Q5):
9609:     - PDET Municipal Plan Analyzer._score_pdet_alignment
9610:     - PDET Municipal Plan Analyzer._score_causal_coherence
9611:     - CDAFFramework._validate_dnp_compliance
9612:     - CDAFFramework._generate_dnp_report
9613:     - IndustrialPolicyProcessor._analyze_causal_dimensions
9614:     - AdaptivePriorCalculator.validate_quality_criteria
9615:     """
9616:
9617:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9618:         raw_evidence = {}
9619:
9620:         # Step 1: Score PDET alignment
9621:         pdet_alignment = self._execute_method(
9622:             "PDET Municipal Plan Analyzer", "_score_pdet_alignment", context
9623:         )
9624:
9625:         # Step 2: Score causal coherence
9626:         causal_coherence = self._execute_method(
9627:             "PDET Municipal Plan Analyzer", "_score_causal_coherence", context

```

```
9628:     )
9629:
9630:     # Step 3: Validate DNP compliance
9631:     dnp_compliance = self._execute_method(
9632:         "CDAFFramework", "_validate_dnp_compliance", context
9633:     )
9634:     dnp_report = self._execute_method(
9635:         "CDAFFramework", "_generate_dnp_report", context,
9636:         compliance=dnp_compliance
9637:     )
9638:
9639:     # Step 4: Analyze causal dimensions
9640:     causal_dimensions = self._execute_method(
9641:         "IndustrialPolicyProcessor", "_analyze_causal_dimensions", context
9642:     )
9643:
9644:     # Step 5: Validate quality criteria
9645:     quality_validation = self._execute_method(
9646:         "AdaptivePriorCalculator", "validate_quality_criteria", context,
9647:         alignment=pdet_alignment
9648:     )
9649:
9650:     raw_evidence = {
9651:         "pnd_alignment": dnp_compliance,
9652:         "sdg_alignment": context.get("sdg_mappings", []),
9653:         "pdet_alignment": pdet_alignment,
9654:         "alignment_declarations": dnp_report.get("declarations", []),
9655:         "causal_coherence": causal_coherence,
9656:         "causal_dimensions": causal_dimensions,
9657:         "quality_validation": quality_validation,
9658:         "alignment_score": (pdet_alignment.get("score", 0) +
9659:             dnp_compliance.get("score", 0)) / 2
9660:     }
9661:
9662:     return {
9663:         "executor_id": self.executor_id,
9664:         "raw_evidence": raw_evidence,
9665:         "metadata": {
9666:             "methods_executed": [log["method"] for log in self.execution_log],
9667:             "pnd_aligned": dnp_compliance.get("is_compliant", False),
9668:             "sdgs_referenced": len(context.get("sdg_mappings", []))
9669:         },
9670:         "execution_metrics": {
9671:             "methods_count": len(self.execution_log),
9672:             "all_succeeded": all(log["success"] for log in self.execution_log)
9673:         }
9674:     }
9675:
9676:
9677: # =====
9678: # DIMENSION 5: IMPACTS
9679: # =====
9680:
9681: class D5_Q1_LongTermVisionAnalyzer(BaseExecutor):
9682:     """
9683:         Analyzes long-term impacts, transmission routes, and time lags.
```

```
9684:  
9685:     Methods (from D5-Q1):  
9686:         - PDETMunicipalPlanAnalyzer.generate_counterfactuals  
9687:         - PDETMunicipalPlanAnalyzer._simulate_intervention  
9688:         - PDETMunicipalPlanAnalyzer._generate_scenario_narrative  
9689:         - PDETMunicipalPlanAnalyzer._find_mediator_mentions  
9690:         - TeoriaCambio._validar orden causal  
9691:         - CausalExtractor._assess_temporal_coherence  
9692:         - TextMiningEngine._generate_interventions  
9693:         - BayesianCounterfactualAuditor.construct_scm  
9694:     """  
9695:  
9696:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:  
9697:         raw_evidence = {}  
9698:  
9699:         # Step 1: Generate counterfactuals  
9700:         counterfactuals = self._execute_method(  
9701:             "PDETMunicipalPlanAnalyzer", "generate_counterfactuals", context  
9702:         )  
9703:  
9704:         # Step 2: Simulate interventions  
9705:         simulation = self._execute_method(  
9706:             "PDETMunicipalPlanAnalyzer", "_simulate_intervention", context,  
9707:             counterfactuals=counterfactuals  
9708:         )  
9709:  
9710:         # Step 3: Generate scenario narratives  
9711:         scenarios = self._execute_method(  
9712:             "PDETMunicipalPlanAnalyzer", "_generate_scenario_narrative", context,  
9713:             simulation=simulation  
9714:         )  
9715:  
9716:         # Step 4: Find mediator mentions  
9717:         mediators = self._execute_method(  
9718:             "PDETMunicipalPlanAnalyzer", "_find_mediator_mentions", context  
9719:         )  
9720:  
9721:         # Step 5: Validate causal order  
9722:         causal_order = self._execute_method(  
9723:             "TeoriaCambio", "_validar orden causal", context,  
9724:             mediators=mediators  
9725:         )  
9726:  
9727:         # Step 6: Assess temporal coherence  
9728:         temporal_coherence = self._execute_method(  
9729:             "CausalExtractor", "_assess_temporal_coherence", context  
9730:         )  
9731:  
9732:         # Step 7: Generate interventions  
9733:         interventions = self._execute_method(  
9734:             "TextMiningEngine", "_generate_interventions", context  
9735:         )  
9736:  
9737:         # Step 8: Construct SCM  
9738:         scm = self._execute_method(  
9739:             "BayesianCounterfactualAuditor", "construct_scm", context,
```

```
9740:         order=causal_order
9741:     )
9742:
9743:     raw_evidence = {
9744:         "long_term_impacts": context.get("impact_indicators", []),
9745:         "structural_transformations": scenarios,
9746:         "transmission_routes": mediators,
9747:         "expected_time_lags": temporal_coherence.get("time_lags", []),
9748:         "counterfactual_analysis": counterfactuals,
9749:         "simulation_results": simulation,
9750:         "causal_pathways": scm,
9751:         "intervention_scenarios": interventions
9752:     }
9753:
9754:     return {
9755:         "executor_id": self.executor_id,
9756:         "raw_evidence": raw_evidence,
9757:         "metadata": {
9758:             "methods_executed": [log["method"] for log in self.execution_log],
9759:             "impacts_defined": len(context.get("impact_indicators", [])),
9760:             "mediators_identified": len(mediators)
9761:         },
9762:         "execution_metrics": {
9763:             "methods_count": len(self.execution_log),
9764:             "all_succeeded": all(log["success"] for log in self.execution_log)
9765:         }
9766:     }
9767:
9768:
9769: class D5_Q2_CompositeMeasurementValidator(BaseExecutor):
9770: """
9771:     DIM05_Q02_COMPOSITE_PROXY_VALIDITY à\200\224 Validates composite indices/proxies for complex impacts (canonical D5).
9772:     Epistemic mix: statistical robustness (E-value), Bayesian confidence, normative reporting quality, and semantic consistency.
9773:
9774:     Methods (from D5-Q2):
9775:     - PDET Municipal Plan Analyzer._quality_to_dict
9776:     - PolicyAnalysisEmbedder.process_document
9777:     - PolicyAnalysisEmbedder._filter_by_pdq
9778:     - PolicyAnalysisEmbedder._extract_numerical_values
9779:     - PolicyAnalysisEmbedder._compute_overall_confidence
9780:     - PolicyAnalysisEmbedder._embed_texts
9781:     - PolicyTextProcessor.normalize_unicode
9782:     - PolicyTextProcessor.segment_into_sentences
9783:     - PolicyTextProcessor.compile_pattern
9784:     - PolicyTextProcessor.extract_contextual_window
9785:     - IndustrialPolicyProcessor._compute_evidence_confidence
9786:     - IndustrialPolicyProcessor._compute_avg_confidence
9787:     - IndustrialPolicyProcessor._construct_evidence_bundle
9788:     - PDET Municipal Plan Analyzer.generate_executive_report
9789:     - BayesianCounterfactualAuditor.aggregate_risk_and_prioritize
9790:     - PDET Municipal Plan Analyzer._interpret_sensitivity
9791:     - PDET Municipal Plan Analyzer._interpret_overall_quality
9792:     - PolicyAnalysisEmbedder.get_diagnostics
9793:     - PDET Municipal Plan Analyzer.calculate_quality_score
9794:     - PDET Municipal Plan Analyzer._estimate_score_confidence
9795:     - PDET Municipal Plan Analyzer._compute_e_value
```

```
9796:     - PDETMunicipalPlanAnalyzer._compute_robustness_value
9797:     - ReportingEngine._calculate_quality_score
9798:     - BayesianMechanismInference._aggregate_bayesian_confidence
9799:     - PolicyAnalysisEmbedder.evaluate_policy_numerical_consistency
9800:     - FinancialAuditor._calculate_sufficiency
9801:     """
9802:
9803:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
9804:         raw_evidence = {}
9805:         dim_info = get_dimension_info(CanonicalDimension.D5.value)
9806:         document_text = context.get("document_text", "")
9807:         document_metadata = context.get("metadata", {})
9808:
9809:         # Step 1: Calculate quality scores
9810:         quality_score = self._execute_method(
9811:             "PDETMunicipalPlanAnalyzer", "calculate_quality_score", context
9812:         )
9813:         score_confidence = self._execute_method(
9814:             "PDETMunicipalPlanAnalyzer", "_estimate_score_confidence", context,
9815:             score=quality_score
9816:         )
9817:
9818:         # Step 2: Compute robustness metrics
9819:         e_value = self._execute_method(
9820:             "PDETMunicipalPlanAnalyzer", "_compute_e_value", context,
9821:             score=quality_score
9822:         )
9823:         robustness = self._execute_method(
9824:             "PDETMunicipalPlanAnalyzer", "_compute_robustness_value", context,
9825:             score=quality_score
9826:         )
9827:         sensitivity_interpretation = self._execute_method(
9828:             "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
9829:             e_value=e_value,
9830:             robustness=robustness
9831:         )
9832:
9833:         # Step 3: Calculate reporting quality score
9834:         reporting_quality = self._execute_method(
9835:             "ReportingEngine", "calculate_quality_score", context
9836:         )
9837:
9838:         # Step 4: Aggregate Bayesian confidence
9839:         bayesian_confidence = self._execute_method(
9840:             "BayesianMechanismInference", "aggregate_bayesian_confidence", context,
9841:             scores=[quality_score, reporting_quality]
9842:         )
9843:
9844:         # Step 5: Evaluate numerical consistency
9845:         numerical_consistency = self._execute_method(
9846:             "PolicyAnalysisEmbedder", "evaluate_policy_numerical_consistency", context
9847:         )
9848:         embedder_diagnostics = self._execute_method(
9849:             "PolicyAnalysisEmbedder", "get_diagnostics", context
9850:         )
9851:         processed_chunks = self._execute_method(
```

```
9852:         "PolicyAnalysisEmbedder", "process_document", context,
9853:         document_text=document_text,
9854:         document_metadata=document_metadata
9855:     )
9856:     pdq_filter = self._execute_method(
9857:         "PolicyAnalysisEmbedder", "_generate_query_from_pdq", context,
9858:         pdq={"policy": context.get("policy_area"), "dimension": dim_info.code}
9859:     )
9860:     filtered_chunks = self._execute_method(
9861:         "PolicyAnalysisEmbedder", "_filter_by_pdq", context,
9862:         chunks=processed_chunks,
9863:         pdq_filter=pdq_filter
9864:     )
9865:     numerical_values = self._execute_method(
9866:         "PolicyAnalysisEmbedder", "_extract_numerical_values", context,
9867:         chunks=processed_chunks
9868:     )
9869:     embedded_texts = self._execute_method(
9870:         "PolicyAnalysisEmbedder", "_embed_texts", context,
9871:         texts=[c.get("content", "") for c in processed_chunks] if isinstance(processed_chunks, list) else []
9872:     )
9873:     overall_confidence = self._execute_method(
9874:         "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
9875:         relevant_chunks=filtered_chunks[:5] if isinstance(filtered_chunks, list) else [],
9876:         numerical_eval=bayesian_confidence if isinstance(bayesian_confidence, dict) else {"evidence_strength": "weak", "numerical_coherence": 0.0}
9877:     )
9878:
9879:     # Step 6: Calculate sufficiency
9880:     sufficiency = self._execute_method(
9881:         "FinancialAuditor", "_calculate_sufficiency", context
9882:     )
9883:     overall_interpretation = self._execute_method(
9884:         "PDET MunicipalPlanAnalyzer", "_interpret_overall_quality", context,
9885:         score=getattr(quality_score, "overall_score", quality_score)
9886:     )
9887:     risk_prioritization = self._execute_method(
9888:         "BayesianCounterfactualAuditor", "aggregate_risk_and_prioritize", context,
9889:         omission_score=1 - quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.2,
9890:         insufficiency_score=1 - sufficiency.get("coverage_ratio", 0.0),
9891:         unnecessity_score=1 - (robustness if isinstance(robustness, (int, float)) else 0.0),
9892:         causal_effect=e_value,
9893:         feasibility=quality_score.financial_feasibility if hasattr(quality_score, "financial_feasibility") else 0.8,
9894:         cost=1.0
9895:     )
9896:     normalized_text = self._execute_method(
9897:         "PolicyTextProcessor", "normalize_unicode", context,
9898:         text=document_text
9899:     )
9900:     segmented_sentences = self._execute_method(
9901:         "PolicyTextProcessor", "segment_into_sentences", context,
9902:         text=document_text
9903:     )
9904:     evidence_confidence = self._execute_method(
9905:         "IndustrialPolicyProcessor", "_compute_evidence_confidence", context,
9906:         matches=context.get("proxy_indicators", []),
9907:         text_length=len(document_text),
```

```

9908:         pattern_specificity=0.5
9909:     )
9910:     avg_confidence = self._execute_method(
9911:         "IndustrialPolicyProcessor", "_compute_avg_confidence", context,
9912:         dimension_analysis={"D5": {"dimension_confidence": bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict)
else 0.0}}
9913:     )
9914:     quality_dict = self._execute_method(
9915:         "PDETMunicipalPlanAnalyzer", "_quality_to_dict", context,
9916:         quality=quality_score
9917:     )
9918:     evidence_bundle = self._execute_method(
9919:         "IndustrialPolicyProcessor", "_construct_evidence_bundle", context,
9920:         dimension=None,
9921:         category="composite",
9922:         matches=context.get("proxy_indicators", []),
9923:         positions=[],
9924:         confidence=bayesian_confidence.get("numerical_coherence", 0.0) if isinstance(bayesian_confidence, dict) else 0.0
9925:     )
9926:     compiled_pattern = self._execute_method(
9927:         "PolicyTextProcessor", "compile_pattern", context,
9928:         pattern_str=r"[A-Z]{2,}\s+\d+"
9929:     )
9930:     contextual_window = self._execute_method(
9931:         "PolicyTextProcessor", "extract_contextual_window", context,
9932:         text=document_text,
9933:         match_position=0,
9934:         window_size=200
9935:     )
9936:     exec_report = self._execute_method(
9937:         "PDETMunicipalPlanAnalyzer", "generate_executive_report", context,
9938:         analysis_results={"quality_score": quality_dict, "financial_analysis": context.get("financial_analysis", {})} or {"total_budget": 0, "funding_sou
rces": {}, "confidence": (0, 0)}
9939:     )
9940:     export_result = self._execute_method(
9941:         "IndustrialPolicyProcessor", "export_results", context,
9942:         results={"quality": quality_dict, "robustness": robustness},
9943:         output_path="output/composite_results.json"
9944:     )
9945:
9946:     raw_evidence = {
9947:         "composite_indices": context.get("composite_indicators", []),
9948:         "proxy_indicators": context.get("proxy_indicators", []),
9949:         "validity_justification": score_confidence,
9950:         "robustness_metrics": {
9951:             "e_value": e_value,
9952:             "robustness": robustness,
9953:             "interpretation": sensitivity_interpretation
9954:         },
9955:         "quality_scores": {
9956:             "overall": quality_score,
9957:             "reporting": reporting_quality
9958:         },
9959:         "bayesian_confidence": bayesian_confidence,
9960:         "numerical_consistency": numerical_consistency,
9961:         "measurement_sufficiency": sufficiency,

```

```
9962:         "embedder_diagnostics": embedder_diagnostics,
9963:         "quality_interpretation": overall_interpretation,
9964:         "pdq_filter": pdq_filter,
9965:         "filtered_chunks": filtered_chunks,
9966:         "numerical_values": numerical_values,
9967:         "embedded_texts": embedded_texts,
9968:         "overall_confidence": overall_confidence,
9969:         "risk_prioritization": risk_prioritization,
9970:         "normalized_text": normalized_text,
9971:         "segmented_sentences": segmented_sentences,
9972:         "evidence_confidence": evidence_confidence,
9973:         "avg_confidence": avg_confidence,
9974:         "quality_dict": quality_dict,
9975:         "compiled_pattern": compiled_pattern,
9976:         "contextual_window": contextual_window,
9977:         "evidence_bundle": evidence_bundle,
9978:         "executive_report": exec_report,
9979:         "export_result": export_result
9980:     }
9981:
9982:     return {
9983:         "executor_id": self.executor_id,
9984:         "raw_evidence": raw_evidence,
9985:         "metadata": {
9986:             "methods_executed": [log["method"] for log in self.execution_log],
9987:             "composite_indices_count": len(context.get("composite_indicators", [])),
9988:             "validity_score": score_confidence,
9989:             "canonical_question": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
9990:             "dimension_code": dim_info.code,
9991:             "dimension_label": dim_info.label
9992:         },
9993:         "execution_metrics": {
9994:             "methods_count": len(self.execution_log),
9995:             "all_succeeded": all(log["success"] for log in self.execution_log)
9996:         }
9997:     }
9998:
9999:
10000: class D5_Q3_IntangibleMeasurementAnalyzer(BaseExecutor):
10001:     """
10002:     Analyzes proxy indicators for intangible impacts with validity documentation.
10003:
10004:     Methods (from D5-Q3):
10005:     - CausalExtractor._calculate_semantic_distance
10006:     - SemanticAnalyzer.extract_semantic_cube
10007:     - BayesianMechanismInference._quantify_uncertainty
10008:     - PDET Municipal Plan Analyzer._find_mediator_mentions
10009:     - PolicyAnalysisEmbedder.get_diagnostics
10010:     - AdaptivePriorCalculator._perturb_evidence
10011:     """
10012:
10013:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10014:         raw_evidence = {}
10015:
10016:         # Step 1: Calculate semantic distance
10017:         semantic_distance = self._execute_method()
```

```
10018:         "CausalExtractor", "_calculate_semantic_distance", context
10019:     )
10020:
10021:     # Step 2: Extract semantic cube
10022:     semantic_cube = self._execute_method(
10023:         "SemanticAnalyzer", "extract_semantic_cube", context
10024:     )
10025:
10026:     # Step 3: Quantify uncertainty
10027:     uncertainty = self._execute_method(
10028:         "BayesianMechanismInference", "_quantify_uncertainty", context,
10029:         semantic_data=semantic_cube
10030:     )
10031:
10032:     # Step 4: Find mediator mentions
10033:     mediators = self._execute_method(
10034:         "PDET MunicipalPlanAnalyzer", "_find_mediator_mentions", context
10035:     )
10036:
10037:     # Step 5: Get diagnostics
10038:     diagnostics = self._execute_method(
10039:         "PolicyAnalysisEmbedder", "get_diagnostics", context,
10040:         mediators=mediators
10041:     )
10042:
10043:     # Step 6: Perturb evidence for sensitivity
10044:     perturbed_evidence = self._execute_method(
10045:         "AdaptivePriorCalculator", "_perturb_evidence", context,
10046:         diagnostics=diagnostics
10047:     )
10048:
10049:     raw_evidence = {
10050:         "intangible_impacts": context.get("intangible_indicators", []),
10051:         "proxy_indicators": context.get("proxy_mappings", []),
10052:         "validity_documentation": diagnostics,
10053:         "limitations_acknowledged": diagnostics.get("limitations", []),
10054:         "semantic_relationships": semantic_cube,
10055:         "semantic_distance": semantic_distance,
10056:         "uncertainty_quantification": uncertainty,
10057:         "sensitivity_to_proxies": perturbed_evidence
10058:     }
10059:
10060:     return {
10061:         "executor_id": self.executor_id,
10062:         "raw_evidence": raw_evidence,
10063:         "metadata": {
10064:             "methods_executed": [log["method"] for log in self.execution_log],
10065:             "intangibles_count": len(context.get("intangible_indicators", [])),
10066:             "proxies_defined": len(context.get("proxy_mappings", []))
10067:         },
10068:         "execution_metrics": {
10069:             "methods_count": len(self.execution_log),
10070:             "all_succeeded": all(log["success"] for log in self.execution_log)
10071:         }
10072:     }
10073:
```

```
10074:
10075: class D5_Q4_SystemicRiskEvaluator(BaseExecutor):
10076:     """
10077:         Evaluates systemic risks that could rupture causal mechanisms.
10078:
10079:     Methods (from D5-Q4):
10080:     - OperationalizationAuditor._audit_systemic_risk
10081:     - BayesianCounterfactualAuditor.refutation_and_sanity_checks
10082:     - BayesianCounterfactualAuditor._test_effect_stability
10083:     - PDETMunicipalPlanAnalyzer._interpret_risk
10084:     - PDETMunicipalPlanAnalyzer._interpret_sensitivity
10085:     - PDETMunicipalPlanAnalyzer._break_cycles
10086:     - AdaptivePriorCalculator.sensitivity_analysis
10087: """
10088:
10089:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10090:         raw_evidence = {}
10091:
10092:         # Step 1: Audit systemic risks
10093:         systemic_risks = self._execute_method(
10094:             "OperationalizationAuditor", "_audit_systemic_risk", context
10095:         )
10096:
10097:         # Step 2: Refutation and sanity checks
10098:         refutation = self._execute_method(
10099:             "BayesianCounterfactualAuditor", "refutation_and_sanity_checks", context,
10100:             risks=systemic_risks
10101:         )
10102:
10103:         # Step 3: Test effect stability
10104:         effect_stability = self._execute_method(
10105:             "BayesianCounterfactualAuditor", "_test_effect_stability", context,
10106:             refutation=refutation
10107:         )
10108:
10109:         # Step 4: Interpret risks
10110:         risk_interpretation = self._execute_method(
10111:             "PDETMunicipalPlanAnalyzer", "_interpret_risk", context,
10112:             risks=systemic_risks
10113:         )
10114:
10115:         # Step 5: Interpret sensitivity
10116:         sensitivity_interpretation = self._execute_method(
10117:             "PDETMunicipalPlanAnalyzer", "_interpret_sensitivity", context,
10118:             stability=effect_stability
10119:         )
10120:
10121:         # Step 6: Break cycles if present
10122:         cycle_breaks = self._execute_method(
10123:             "PDETMunicipalPlanAnalyzer", "_break_cycles", context
10124:         )
10125:
10126:         # Step 7: Sensitivity analysis
10127:         sensitivity = self._execute_method(
10128:             "AdaptivePriorCalculator", "sensitivity_analysis", context,
10129:             risks=systemic_risks
```

```
10130:         )
10131:
10132:     raw_evidence = {
10133:         "macroeconomic_risks": [r for r in systemic_risks if r.get("type") == "macroeconomic"],
10134:         "environmental_risks": [r for r in systemic_risks if r.get("type") == "environmental"],
10135:         "political_risks": [r for r in systemic_risks if r.get("type") == "political"],
10136:         "mechanism_rupture_potential": risk_interpretation.get("rupture_probability", 0),
10137:         "effect_stability": effect_stability,
10138:         "refutation_results": refutation,
10139:         "sensitivity_analysis": sensitivity,
10140:         "cycle_vulnerabilities": cycle_breaks
10141:     }
10142:
10143:     return {
10144:         "executor_id": self.executor_id,
10145:         "raw_evidence": raw_evidence,
10146:         "metadata": {
10147:             "methods_executed": [log["method"] for log in self.execution_log],
10148:             "systemic_risks_identified": len(systemic_risks),
10149:             "high_risk_count": len([r for r in systemic_risks if r.get("severity") == "high"])
10150:         },
10151:         "execution_metrics": {
10152:             "methods_count": len(self.execution_log),
10153:             "all_succeeded": all(log["success"] for log in self.execution_log)
10154:         }
10155:     }
10156:
10157:
10158: class D5_Q5_RealismAndSideEffectsAnalyzer(BaseExecutor):
10159:     """
10160:     Analyzes realism of impact ambition and potential unintended effects.
10161:
10162:     Methods (from D5-Q5):
10163:     - HierarchicalGenerativeModel.posterior_predictive_check
10164:     - HierarchicalGenerativeModel._ablation_analysis
10165:     - HierarchicalGenerativeModel._calculate_waic_difference
10166:     - AdaptivePriorCalculator._add_ood_noise
10167:     - AdaptivePriorCalculator.validate_quality_criteria
10168:     - PDET MunicipalPlanAnalyzer._compute_e_value
10169:     - PDET MunicipalPlanAnalyzer._compute_robustness_value
10170:     - BayesianMechanismInference._calculate_coherence_factor
10171:     """
10172:
10173:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10174:         raw_evidence = {}
10175:
10176:         # Step 1: Posterior predictive check
10177:         predictive_check = self._execute_method(
10178:             "HierarchicalGenerativeModel", "posterior_predictive_check", context
10179:         )
10180:
10181:         # Step 2: Ablation analysis
10182:         ablation = self._execute_method(
10183:             "HierarchicalGenerativeModel", "_ablation_analysis", context,
10184:             check=predictive_check
10185:         )
```

```
10186:  
10187:     # Step 3: Calculate WAIC difference  
10188:     waic_diff = self._execute_method(  
10189:         "HierarchicalGenerativeModel", "_calculate_waic_difference", context,  
10190:         ablation=ablation  
10191:     )  
10192:  
10193:     # Step 4: Add out-of-distribution noise  
10194:     ood_analysis = self._execute_method(  
10195:         "AdaptivePriorCalculator", "_add_ood_noise", context  
10196:     )  
10197:  
10198:     # Step 5: Validate quality criteria  
10199:     quality_validation = self._execute_method(  
10200:         "AdaptivePriorCalculator", "validate_quality_criteria", context,  
10201:         ood=ood_analysis  
10202:     )  
10203:  
10204:     # Step 6: Compute robustness metrics  
10205:     e_value = self._execute_method(  
10206:         "PDET MunicipalPlanAnalyzer", "_compute_e_value", context  
10207:     )  
10208:     robustness = self._execute_method(  
10209:         "PDET MunicipalPlanAnalyzer", "_compute_robustness_value", context  
10210:     )  
10211:  
10212:     # Step 7: Calculate coherence factor  
10213:     coherence = self._execute_method(  
10214:         "BayesianMechanismInference", "_calculate_coherence_factor", context  
10215:     )  
10216:  
10217:     raw_evidence = {  
10218:         "impact_ambition_level": context.get("declared_ambition", 0),  
10219:         "realism_assessment": predictive_check.get("realism_score", 0),  
10220:         "negative_side_effects": ablation.get("negative_effects", []),  
10221:         "limit_hypotheses": quality_validation.get("limits", []),  
10222:         "robustness_metrics": {  
10223:             "e_value": e_value,  
10224:             "robustness": robustness,  
10225:             "coherence": coherence  
10226:         },  
10227:         "predictive_validity": predictive_check,  
10228:         "ablation_results": ablation,  
10229:         "model_comparison": waic_diff,  
10230:         "ood_sensitivity": ood_analysis  
10231:     }  
10232:  
10233:     return {  
10234:         "executor_id": self.executor_id,  
10235:         "raw_evidence": raw_evidence,  
10236:         "metadata": {  
10237:             "methods_executed": [log["method"] for log in self.execution_log],  
10238:             "realism_score": predictive_check.get("realism_score", 0),  
10239:             "side_effects_identified": len(ablation.get("negative_effects", []))  
10240:         },  
10241:         "execution_metrics": {
```

```
10242:         "methods_count": len(self.execution_log),
10243:         "all_succeeded": all(log["success"] for log in self.execution_log)
10244:     }
10245: }
10246:
10247:
10248: # =====
10249: # DIMENSION 6: CAUSALITY & THEORY OF CHANGE
10250: # =====
10251:
10252: class D6_Q1_ExplicitTheoryBuilder(BaseExecutor):
10253:     """
10254:     Builds/validates explicit Theory of Change with diagram and assumptions.
10255:
10256:     Methods (from D6-Q1):
10257:     - TeoriaCambio.construir_grafo_causal
10258:     - TeoriaCambio.validacion_completa
10259:     - TeoriaCambio.export_nodes
10260:     - ReportingEngine.generate_causal_diagram
10261:     - ReportingEngine.generate_causal_model_json
10262:     - AdvancedDAGValidator.export_nodes
10263:     - PDET MunicipalPlanAnalyzer.export_causal_network
10264:     - CausalExtractor.extract_causal_hierarchy
10265:     """
10266:
10267:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10268:         raw_evidence = {}
10269:
10270:         # Step 1: Build causal graph
10271:         causal_graph = self._execute_method(
10272:             "TeoriaCambio", "construir_grafo_causal", context
10273:         )
10274:
10275:         # Step 2: Complete validation
10276:         validation = self._execute_method(
10277:             "TeoriaCambio", "validacion_completa", context,
10278:             graph=causal_graph
10279:         )
10280:
10281:         # Step 3: Export nodes from Theory of Change
10282:         toc_nodes = self._execute_method(
10283:             "TeoriaCambio", "export_nodes", context,
10284:             graph=causal_graph
10285:         )
10286:
10287:         # Step 4: Generate causal diagram
10288:         diagram = self._execute_method(
10289:             "ReportingEngine", "generate_causal_diagram", context,
10290:             graph=causal_graph
10291:         )
10292:
10293:         # Step 5: Generate causal model JSON
10294:         model_json = self._execute_method(
10295:             "ReportingEngine", "generate_causal_model_json", context,
10296:             graph=causal_graph
10297:         )
```

```

10298:
10299:     # Step 6: Export nodes from DAG validator
10300:     dag_nodes = self._execute_method(
10301:         "AdvancedDAGValidator", "export_nodes", context,
10302:         graph=causal_graph
10303:     )
10304:
10305:     # Step 7: Export causal network
10306:     network_export = self._execute_method(
10307:         "PDET MunicipalPlanAnalyzer", "export_causal_network", context,
10308:         graph=causal_graph
10309:     )
10310:
10311:     # Step 8: Extract causal hierarchy
10312:     hierarchy = self._execute_method(
10313:         "CausalExtractor", "extract_causal_hierarchy", context
10314:     )
10315:
10316:     raw_evidence = {
10317:         "toc_exists": len(causal_graph) > 0,
10318:         "toc_diagram": diagram,
10319:         "toc_json": model_json,
10320:         "causal_graph": causal_graph,
10321:         "nodes": toc_nodes,
10322:         "causes_identified": hierarchy.get("causes", []),
10323:         "mediators_identified": hierarchy.get("mediators", []),
10324:         "assumptions": validation.get("assumptions", []),
10325:         "network_structure": network_export,
10326:         "validation_results": validation
10327:     }
10328:
10329:     return {
10330:         "executor_id": self.executor_id,
10331:         "raw_evidence": raw_evidence,
10332:         "metadata": {
10333:             "methods_executed": [log["method"] for log in self.execution_log],
10334:             "nodes_count": len(toc_nodes),
10335:             "assumptions_count": len(validation.get("assumptions", []))
10336:         },
10337:         "execution_metrics": {
10338:             "methods_count": len(self.execution_log),
10339:             "all_succeeded": all(log["success"] for log in self.execution_log)
10340:         }
10341:     }
10342:
10343:
10344: class D6_Q2_LogicalProportionalityValidator(BaseExecutor):
10345:     """
10346:     Validates logical proportionality: no leaps, intervention matches result scale.
10347:
10348:     Methods (from D6-Q2):
10349:     - BeachEvidentialTest.apply_test_logic
10350:     - BayesianMechanismInference._test_necessity
10351:     - BayesianMechanismInference._test_sufficiency
10352:     - BayesianMechanismInference._calculate_coherence_factor
10353:     - BayesianCounterfactualAuditor._test_effect_stability

```

```
10354:     - IndustrialGradeValidator.validate_connection_matrix
10355:     - PolicyAnalysisEmbedder._compute_overall_confidence
10356:     """
10357:
10358:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10359:         raw_evidence = {}
10360:
10361:         # Step 1: Apply evidential tests
10362:         evidential_tests = self._execute_method(
10363:             "BeachEvidentialTest", "apply_test_logic", context
10364:         )
10365:
10366:         # Step 2: Test necessity
10367:         necessity_test = self._execute_method(
10368:             "BayesianMechanismInference", "_test_necessity", context
10369:         )
10370:
10371:         # Step 3: Test sufficiency
10372:         sufficiency_test = self._execute_method(
10373:             "BayesianMechanismInference", "_test_sufficiency", context
10374:         )
10375:
10376:         # Step 4: Calculate coherence factor
10377:         coherence_factor = self._execute_method(
10378:             "BayesianMechanismInference", "_calculate_coherence_factor", context,
10379:             necessity=necessity_test,
10380:             sufficiency=sufficiency_test
10381:         )
10382:
10383:         # Step 5: Test effect stability
10384:         effect_stability = self._execute_method(
10385:             "BayesianCounterfactualAuditor", "_test_effect_stability", context
10386:         )
10387:
10388:         # Step 6: Validate connection matrix
10389:         connection_validation = self._execute_method(
10390:             "IndustrialGradeValidator", "validate_connection_matrix", context
10391:         )
10392:
10393:         # Step 7: Compute overall confidence
10394:         overall_confidence = self._execute_method(
10395:             "PolicyAnalysisEmbedder", "_compute_overall_confidence", context,
10396:             tests=[necessity_test, sufficiency_test, effect_stability]
10397:         )
10398:
10399:         raw_evidence = {
10400:             "logical_leaps_detected": evidential_tests.get("leaps", []),
10401:             "intervention_scale": context.get("intervention_magnitude", 0),
10402:             "result_scale": context.get("result_magnitude", 0),
10403:             "proportionality_ratio": context.get("intervention_magnitude", 0) / max(context.get("result_magnitude", 1), 1),
10404:             "necessity_score": necessity_test,
10405:             "sufficiency_score": sufficiency_test,
10406:             "coherence_factor": coherence_factor,
10407:             "effect_stability": effect_stability,
10408:             "connection_validation": connection_validation,
10409:             "overall_confidence": overall_confidence,
```

```
10410:         "implementation_miracles": [leap for leap in evidential_tests.get("leaps", [])
10411:                                         if leap.get("type") == "miracle"]
10412:     }
10413:
10414:     return {
10415:         "executor_id": self.executor_id,
10416:         "raw_evidence": raw_evidence,
10417:         "metadata": {
10418:             "methods_executed": [log["method"] for log in self.execution_log],
10419:             "leaps_detected": len(evidential_tests.get("leaps", [])),
10420:             "proportionality_adequate": abs(raw_evidence["proportionality_ratio"] - 1.0) < 0.5
10421:         },
10422:         "execution_metrics": {
10423:             "methods_count": len(self.execution_log),
10424:             "all_succeeded": all(log["success"] for log in self.execution_log)
10425:         }
10426:     }
10427:
10428:
10429: class D6_Q3_ValidationTestingAnalyzer(BaseExecutor):
10430: """
10431:     Analyzes validation/testing proposals for weak assumptions before scaling.
10432:
10433:     Methods (from D6-Q3):
10434:     - IndustrialGradeValidator.execute_suite
10435:     - IndustrialGradeValidator.validate_engine_readiness
10436:     - IndustrialGradeValidator._benchmark_operation
10437:     - AdaptivePriorCalculator.validate_quality_criteria
10438:     - HierarchicalGenerativeModel._calculate_r_hat
10439:     - HierarchicalGenerativeModel._calculate_ess
10440:     - AdvancedDAGValidator.calculate_acyclicity_pvalue
10441:     - PerformanceAnalyzer.analyze_performance
10442: """
10443:
10444: def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10445:     raw_evidence = {}
10446:
10447:     # Step 1: Execute validation suite
10448:     validation_suite = self._execute_method(
10449:         "IndustrialGradeValidator", "execute_suite", context
10450:     )
10451:
10452:     # Step 2: Validate engine readiness
10453:     readiness = self._execute_method(
10454:         "IndustrialGradeValidator", "validate_engine_readiness", context
10455:     )
10456:
10457:     # Step 3: Benchmark operations
10458:     benchmarks = self._execute_method(
10459:         "IndustrialGradeValidator", "_benchmark_operation", context
10460:     )
10461:
10462:     # Step 4: Validate quality criteria
10463:     quality_validation = self._execute_method(
10464:         "AdaptivePriorCalculator", "validate_quality_criteria", context
10465:     )
```

```
10466:  
10467:     # Step 5: Calculate convergence diagnostics  
10468:     r_hat = self._execute_method(  
10469:         "HierarchicalGenerativeModel", "_calculate_r_hat", context  
10470:     )  
10471:     ess = self._execute_method(  
10472:         "HierarchicalGenerativeModel", "_calculate_ess", context  
10473:     )  
10474:  
10475:     # Step 6: Calculate acyclicity p-value  
10476:     acyclicity_p = self._execute_method(  
10477:         "AdvancedDAGValidator", "calculate_acyclicity_pvalue", context  
10478:     )  
10479:  
10480:     # Step 7: Analyze performance  
10481:     performance = self._execute_method(  
10482:         "PerformanceAnalyzer", "analyze_performance", context  
10483:     )  
10484:  
10485:     raw_evidence = {  
10486:         "inconsistencies_recognized": validation_suite.get("inconsistencies", []),  
10487:         "weak_assumptions": quality_validation.get("weak_assumptions", []),  
10488:         "pilot_proposals": context.get("pilot_programs", []),  
10489:         "testing_proposals": context.get("testing_plans", []),  
10490:         "validation_before_scaling": readiness.get("ready_to_scale", False),  
10491:         "validation_results": validation_suite,  
10492:         "quality_criteria": quality_validation,  
10493:         "convergence_diagnostics": {  
10494:             "r_hat": r_hat,  
10495:             "ess": ess,  
10496:             "acyclicity_p": acyclicity_p  
10497:         },  
10498:         "performance_analysis": performance,  
10499:         "benchmarks": benchmarks  
10500:     }  
10501:  
10502:     return {  
10503:         "executor_id": self.executor_id,  
10504:         "raw_evidence": raw_evidence,  
10505:         "metadata": {  
10506:             "methods_executed": [log["method"] for log in self.execution_log],  
10507:             "inconsistencies_count": len(validation_suite.get("inconsistencies", [])),  
10508:             "pilots_proposed": len(context.get("pilot_programs", []))  
10509:         },  
10510:         "execution_metrics": {  
10511:             "methods_count": len(self.execution_log),  
10512:             "all_succeeded": all(log["success"] for log in self.execution_log)  
10513:         }  
10514:     }  
10515:  
10516:  
10517: class D6_Q4_FeedbackLoopAnalyzer(BaseExecutor):  
10518:     """  
10519:         Analyzes monitoring system with correction mechanisms and learning processes.  
10520:  
10521:     Methods (from D6-Q4):
```

```
10522:     - ConfigLoader.update_priors_from_feedback
10523:     - ConfigLoader.check_uncertainty_reduction_criterion
10524:     - ConfigLoader._save_prior_history
10525:     - ConfigLoader._load_uncertainty_history
10526:     - CDAFFramework._extract_feedback_from_audit
10527:     - AdvancedDAGValidator._calculate_node_importance
10528:     - BayesFactorTable.get_bayes_factor
10529:     """
10530:
10531:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10532:         raw_evidence = {}
10533:
10534:         # Step 1: Update priors from feedback
10535:         prior_updates = self._execute_method(
10536:             "ConfigLoader", "update_priors_from_feedback", context
10537:         )
10538:
10539:         # Step 2: Check uncertainty reduction
10540:         uncertainty_reduction = self._execute_method(
10541:             "ConfigLoader", "check_uncertainty_reduction_criterion", context,
10542:             updates=prior_updates
10543:         )
10544:
10545:         # Step 3: Save prior history
10546:         history_saved = self._execute_method(
10547:             "ConfigLoader", "_save_prior_history", context,
10548:             updates=prior_updates
10549:         )
10550:
10551:         # Step 4: Load uncertainty history
10552:         uncertainty_history = self._execute_method(
10553:             "ConfigLoader", "_load_uncertainty_history", context
10554:         )
10555:
10556:         # Step 5: Extract feedback from audit
10557:         feedback_extracted = self._execute_method(
10558:             "CDAFFramework", "_extract_feedback_from_audit", context
10559:         )
10560:
10561:         # Step 6: Calculate node importance
10562:         node_importance = self._execute_method(
10563:             "AdvancedDAGValidator", "_calculate_node_importance", context
10564:         )
10565:
10566:         # Step 7: Get Bayes factor
10567:         bayes_factor = self._execute_method(
10568:             "BayesFactorTable", "get_bayes_factor", context,
10569:             updates=prior_updates
10570:         )
10571:
10572:         raw_evidence = {
10573:             "monitoring_system_described": len(context.get("monitoring_indicators", [])) > 0,
10574:             "correction_mechanisms": feedback_extracted.get("mechanisms", []),
10575:             "feedback_loops": feedback_extracted.get("loops", []),
10576:             "learning_processes": feedback_extracted.get("learning", []),
10577:             "prior_updates": prior_updates,
```

```
10578:         "uncertainty_reduction": uncertainty_reduction,
10579:         "uncertainty_history": uncertainty_history,
10580:         "node_importance": node_importance,
10581:         "learning_strength": bayes_factor
10582:     }
10583:
10584:     return {
10585:         "executor_id": self.executor_id,
10586:         "raw_evidence": raw_evidence,
10587:         "metadata": {
10588:             "methods_executed": [log["method"] for log in self.execution_log],
10589:             "feedback_mechanisms": len(feedback_extracted.get("mechanisms", [])),
10590:             "learning_processes": len(feedback_extracted.get("learning", []))
10591:         },
10592:         "execution_metrics": {
10593:             "methods_count": len(self.execution_log),
10594:             "all_succeeded": all(log["success"] for log in self.execution_log)
10595:         }
10596:     }
10597:
10598:
10599: class D6_Q5_ContextualAdaptabilityEvaluator(BaseExecutor):
10600:     """
10601:     Evaluates contextual adaptation: differential impacts and territorial constraints.
10602:
10603:     Methods (from D6-Q5):
10604:     - CausalExtractor._calculate_language_specificity
10605:     - CausalExtractor._assess_temporal_coherence
10606:     - TextMiningEngine.diagnose_critical_links
10607:     - CausalInferenceSetup.identify_failure_points
10608:     - CausalInferenceSetup._get_dynamics_pattern
10609:     - SemanticProcessor.chunk_text
10610:     - SemanticProcessor._detect_pdm_structure
10611:     - SemanticProcessor._detect_table
10612:     - AdaptivePriorCalculator.generate_traceability_record
10613:     """
10614:
10615:     def execute(self, context: Dict[str, Any]) -> Dict[str, Any]:
10616:         raw_evidence = {}
10617:
10618:         # Step 1: Calculate language specificity
10619:         language_specificity = self._execute_method(
10620:             "CausalExtractor", "_calculate_language_specificity", context
10621:         )
10622:
10623:         # Step 2: Assess temporal coherence
10624:         temporal_coherence = self._execute_method(
10625:             "CausalExtractor", "_assess_temporal_coherence", context
10626:         )
10627:
10628:         # Step 3: Diagnose critical links
10629:         critical_links = self._execute_method(
10630:             "TextMiningEngine", "diagnose_critical_links", context
10631:         )
10632:
10633:         # Step 4: Identify failure points
```

```
10634:         failure_points = self._execute_method(
10635:             "CausalInferenceSetup", "identify_failure_points", context
10636:         )
10637:
10638:         # Step 5: Get dynamics pattern
10639:         dynamics_pattern = self._execute_method(
10640:             "CausalInferenceSetup", "_get_dynamics_pattern", context
10641:         )
10642:
10643:         # Step 6: Process text structure
10644:         text_chunks = self._execute_method(
10645:             "SemanticProcessor", "chunk_text", context
10646:         )
10647:         pdm_structure = self._execute_method(
10648:             "SemanticProcessor", "_detect_pdm_structure", context,
10649:             chunks=text_chunks
10650:         )
10651:         table_detection = self._execute_method(
10652:             "SemanticProcessor", "_detect_table", context,
10653:             chunks=text_chunks
10654:         )
10655:
10656:         # Step 7: Generate traceability record
10657:         traceability = self._execute_method(
10658:             "AdaptivePriorCalculator", "generate_traceability_record", context,
10659:             specificity=language_specificity
10660:         )
10661:
10662:         raw_evidence = {
10663:             "context_adaptation": language_specificity.get("adaptation_level", 0),
10664:             "differential_impacts_recognized": critical_links.get("differential_groups", []),
10665:             "specific_groups_mentioned": critical_links.get("target_groups", []),
10666:             "territorial_constraints": failure_points.get("territorial", []),
10667:             "local_context_integration": pdm_structure.get("local_sections", []),
10668:             "language_specificity": language_specificity,
10669:             "temporal_coherence": temporal_coherence,
10670:             "dynamics_pattern": dynamics_pattern,
10671:             "structure_analysis": pdm_structure,
10672:             "traceability": traceability
10673:         }
10674:
10675:         return {
10676:             "executor_id": self.executor_id,
10677:             "raw_evidence": raw_evidence,
10678:             "metadata": {
10679:                 "methods_executed": [log["method"] for log in self.execution_log],
10680:                 "groups_identified": len(critical_links.get("target_groups", [])),
10681:                 "territorial_constraints": len(failure_points.get("territorial", []))
10682:             },
10683:             "execution_metrics": {
10684:                 "methods_count": len(self.execution_log),
10685:                 "all_succeeded": all(log["success"] for log in self.execution_log)
10686:             }
10687:         }
10688:
10689:
```

```
10690: # =====
10691: # EXECUTOR REGISTRY
10692: # =====
10693:
10694: EXECUTOR_REGISTRY = {
10695:     "D1-Q1": D1_Q1_QuantitativeBaselineExtractor,
10696:     "D1-Q2": D1_Q2_ProblemDimensioningAnalyzer,
10697:     "D1-Q3": D1_Q3_BudgetAllocationTracer,
10698:     "D1-Q4": D1_Q4_InstitutionalCapacityIdentifier,
10699:     "D1-Q5": D1_Q5_ScopeJustificationValidator,
10700:
10701:     "D2-Q1": D2_Q1_StructuredPlanningValidator,
10702:     "D2-Q2": D2_Q2_InterventionLogicInferencer,
10703:     "D2-Q3": D2_Q3_RootCauseLinkageAnalyzer,
10704:     "D2-Q4": D2_Q4_RiskManagementAnalyzer,
10705:     "D2-Q5": D2_Q5_StrategicCoherenceEvaluator,
10706:
10707:     "D3-Q1": D3_Q1_IndicatorQualityValidator,
10708:     "D3-Q2": D3_Q2_TargetProportionalityAnalyzer,
10709:     "D3-Q3": D3_Q3_TraceabilityValidator,
10710:     "D3-Q4": D3_Q4_TechnicalFeasibilityEvaluator,
10711:     "D3-Q5": D3_Q5_OutputOutcomeLinkageAnalyzer,
10712:
10713:     "D4-Q1": D4_Q1_OutcomeMetricsValidator,
10714:     "D4-Q2": D4_Q2_CausalChainValidator,
10715:     "D4-Q3": D4_Q3_AmbitionJustificationAnalyzer,
10716:     "D4-Q4": D4_Q4_ProblemSolvencyEvaluator,
10717:     "D4-Q5": D4_Q5_VerticalAlignmentValidator,
10718:
10719:     "D5-Q1": D5_Q1_LongTermVisionAnalyzer,
10720:     "D5-Q2": D5_Q2_CompositeMeasurementValidator,
10721:     "D5-Q3": D5_Q3_IntangibleMeasurementAnalyzer,
10722:     "D5-Q4": D5_Q4_SystemicRiskEvaluator,
10723:     "D5-Q5": D5_Q5_RealismAndSideEffectsAnalyzer,
10724:
10725:     "D6-Q1": D6_Q1_ExplicitTheoryBuilder,
10726:     "D6-Q2": D6_Q2_LogicalProportionalityValidator,
10727:     "D6-Q3": D6_Q3_ValidationTestingAnalyzer,
10728:     "D6-Q4": D6_Q4_FeedbackLoopAnalyzer,
10729:     "D6-Q5": D6_Q5_ContextualAdaptabilityEvaluator,
10730: }
10731:
10732:
10733: # =====
10734: # PHASE 2 ORCHESTRATION
10735: # =====
10736:
10737:
10738: def _build_method_executor() -> MethodExecutor:
10739:     """Construct a canonical MethodExecutor via the factory wiring."""
10740:     bundle = build_processor()
10741:     method_executor = getattr(bundle, "method_executor", None)
10742:     if not isinstance(method_executor, MethodExecutor):
10743:         raise RuntimeError("ProcessorBundle did not provide a valid MethodExecutor instance.")
10744:     return method_executor
10745:
```

```
10746:
10747: def _canonical_metadata(executor_id: str) -> Dict[str, Any]:
10748:     """Build canonical metadata block using canonical_notation."""
10749:     metadata: Dict[str, Any] = {}
10750:     try:
10751:         dim_key = executor_id.split("-")[0]
10752:         dim_info = get_dimension_info(dim_key)
10753:         metadata["dimension_code"] = dim_info.code
10754:         metadata["dimension_label"] = dim_info.label
10755:     except Exception:
10756:         pass
10757:
10758:     if executor_id in CANONICAL_QUESTION_LABELS:
10759:         metadata["canonical_question"] = CANONICAL_QUESTION_LABELS[executor_id]
10760:
10761:     return metadata
10762:
10763: def run_phase2_executors(context_package: Dict[str, Any],
10764:                           policy_areas: List[str]) -> Dict[str, Any]:
10765:     """
10766:     Phase 2 Entry Point: Runs all 30 executors for each policy area.
10767:
10768:     Args:
10769:         context_package: Canonical package with document data from Phase 1
10770:         policy_areas: List of policy area identifiers to analyze
10771:
10772:     Returns:
10773:         Dict mapping policy_area -> executor_id -> raw_evidence
10774:     """
10775:     results = {}
10776:     method_executor = _build_method_executor()
10777:
10778:     for policy_area in policy_areas:
10779:         print(f"\n{'='*80}")
10780:         print(f"Processing Policy Area: {policy_area}")
10781:         print(f"{'='*80}\n")
10782:
10783:         # Prepare context for this policy area
10784:         area_context = {
10785:             **context_package,
10786:             "policy_area": policy_area
10787:         }
10788:
10789:         # Execute all 30 executors
10790:         area_results = {}
10791:         for executor_id, executor_class in EXECUTOR_REGISTRY.items():
10792:             print(f"Running {executor_id}: {executor_class.__name__}...")
10793:
10794:             try:
10795:                 # Instantiate executor with config
10796:                 config = load_executor_config(executor_id)
10797:                 executor = executor_class(executor_id, config, method_executor=method_executor)
10798:
10799:                 # Execute and collect results
10800:                 result = executor.execute(area_context)
10801:                 # Append canonical metadata consistently
```

```
10802:         result_metadata = result.get("metadata", {})
10803:         result_metadata.update(_canonical_metadata(executor_id))
10804:         result["metadata"] = result_metadata
10805:         area_results[executor_id] = result
10806:
10807:         print(f"  \u2708\u2723 Success: {len(result['metadata']['methods_executed'])} methods executed")
10808:
10809:     except ExecutorFailure as e:
10810:         print(f"  \u2708\u2723 FAILED: {str(e)}")
10811:         raise # Re-raise to stop execution as per requirement
10812:
10813:     results[policy_area] = area_results
10814:
10815: return results
10816:
10817:
10818: def load_executor_config(executor_id: str) -> Dict[str, Any]:
10819:     """
10820:     Load executor configuration from JSON contract.
10821:
10822:     Args:
10823:         executor_id: Executor identifier (e.g., "D1-Q1")
10824:
10825:     Returns:
10826:         Configuration dictionary from JSON contract
10827:     """
10828:     import json
10829:     from pathlib import Path
10830:
10831:     config_path = Path(f"config/executor_contracts/{executor_id}.json")
10832:
10833:     if not config_path.exists():
10834:         raise FileNotFoundError(f"Executor config not found: {config_path}")
10835:
10836:     with open(config_path, 'r', encoding='utf-8') as f:
10837:         return json.load(f)
10838:
10839:
10840: # =====
10841: # EXAMPLE USAGE
10842: # =====
10843:
10844: if __name__ == "__main__":
10845:     # Example context package from Phase 1
10846:     context_package = {
10847:         "document_path": "data/pdm_municipality_xyz.pdf",
10848:         "document_text": "...", # Full document text
10849:         "tables": [], # Extracted tables from Phase 1
10850:         "embeddings": {}, # Precomputed embeddings
10851:         "entities": [], # Pre-extracted entities
10852:         "metadata": {
10853:             "municipality": "Municipality XYZ",
10854:             "year": 2024,
10855:             "pages": 150
10856:         }
10857:     }
```

```
10858:
10859:     # Policy areas to analyze
10860:     policy_areas = [
10861:         "PA01",    # Education
10862:         "PA02",    # Health
10863:         "PA03",    # Infrastructure
10864:         # ... up to 10+ policy areas
10865:     ]
10866:
10867:     # Run Phase 2
10868:     try:
10869:         results = run_phase2_executors(context_package, policy_areas)
10870:         print("\n" + "="*80)
10871:         print("PHASE 2 COMPLETED SUCCESSFULLY")
10872:         print("="*80)
10873:         print(f"Processed {len(policy_areas)} policy areas")
10874:         print(f"Executed {len(EXECUTOR_REGISTRY)} executors per area")
10875:         print(f"Total executions: {len(policy_areas) * len(EXECUTOR_REGISTRY)}")
10876:
10877:     except ExecutorFailure as e:
10878:         print("\n" + "="*80)
10879:         print("PHASE 2 FAILED")
10880:         print("="*80)
10881:         print(f"Error: {str(e)}")
10882:         print("Execution halted as per requirement: any method failure = executor failure")
10883:
10884:
10885:
10886: =====
10887: FILE: src/farfan_pipeline/core/orchestrator/factory.py
10888: =====
10889:
10890: """
10891: Factory module à\200\224 canonical Dependency Injection (DI) and access control for F.A.R.F.A.N.
10892:
10893: This module is the SINGLE AUTHORITATIVE BOUNDARY for:
10894: - Canonical monolith access (CanonicalQuestionnaire) - loaded ONCE with integrity verification
10895: - Signal registry construction (QuestionnaireSignalRegistry v2.0) from canonical source ONLY
10896: - Method injection via MethodExecutor with signal registry DI
10897: - Orchestrator construction with full DI (questionnaire, method_executor, executor_config)
10898: - EnrichedSignalPack creation and injection per executor (30 executors)
10899: - Hard contracts and validation constants for Phase 1
10900: - SeedRegistry singleton initialization for determinism
10901:
10902: METHOD DISPENSARY PATTERN - Core Architecture:
10903: =====
10904:
10905: The pipeline uses a "method dispensary" pattern where monolithic analyzer classes
10906: serve as "dispensaries" that provide methods to executors. This architecture enables:
10907:
10908: 1. LOOSE COUPLING: Executors orchestrate methods without direct imports
10909: 2. PARTIAL REUSE: Same method used by multiple executors with different contexts
10910: 3. CENTRALIZED MANAGEMENT: All method routing through MethodExecutor with validation
10911: 4. SIGNAL AWARENESS: Methods receive signal packs for pattern matching
10912:
10913: Dispensary Registry (~20 monolith classes, 240+ methods):
```

```
10914: -----
10915: - IndustrialPolicyProcessor (17 methods): Pattern matching, evidence extraction
10916: - PDET MunicipalPlanAnalyzer (52+ methods): LARGEST - financial, causal, entity analysis
10917: - CausalExtractor (28 methods): Goal extraction, causal hierarchy, semantic distance
10918: - FinancialAuditor (13 methods): Budget tracing, allocation gaps, sufficiency
10919: - BayesianMechanismInference (14 methods): Necessity/sufficiency tests, coherence
10920: - BayesianCounterfactualAuditor (9 methods): SCM construction, refutation
10921: - TextMiningEngine (8 methods): Critical link diagnosis, intervention generation
10922: - SemanticAnalyzer (12 methods): Semantic cube, domain classification
10923: - PerformanceAnalyzer (5 methods): Performance metrics, loss functions
10924: - PolicyContradictionDetector (8 methods): Contradiction detection, coherence
10925: - [... 10+ more classes]
10926:
10927: Executor Usage Pattern:
10928: -----
10929: Each of 30 executors uses a UNIQUE COMBINATION of methods:
10930: - D1-Q1 (QuantitativeBaselineExtractor): 17 methods from 9 classes
10931: - D3-Q2 (TargetProportionalityAnalyzer): 24 methods from 7 classes
10932: - D3-Q5 (OutputOutcomeLinkageAnalyzer): 28 methods from 6 classes
10933: - D6-Q3 (ValidationTestingAnalyzer): 8 methods from 4 classes
10934:
10935: Methods are orchestrated via:
10936: '''python
10937: result = self.method_executor.execute(
10938:     class_name="PDET MunicipalPlanAnalyzer",
10939:     method_name="_score_indicators",
10940:     document=doc,
10941:     signal_pack=pack,
10942:     **context
10943: )
10944: '''
10945:
10946: NOT ALL METHODS ARE USED:
10947: - Monoliths contain more methods than executors need
10948: - Only methods in executors_methods.json are actively used
10949: - Phase 1 (ingestion) uses additional methods not in executor contracts
10950: - 14 methods in validation failures (deprecated/private)
10951:
10952: Design Principles (Factory Pattern + DI):
10953: =====
10954:
10955: 1. FACTORY PATTERN: AnalysisPipelineFactory is the ONLY place that instantiates:
10956:   - Orchestrator, MethodExecutor, QuestionnaireSignalRegistry, BaseExecutor instances
10957:   - NO other module should directly instantiate these classes
10958:
10959: 2. DEPENDENCY INJECTION: All components receive dependencies via __init__:
10960:   - Orchestrator receives: questionnaire, method_executor, executor_config, validation_constants
10961:   - MethodExecutor receives: method_registry, arg_router, signal_registry
10962:   - BaseExecutor (30 classes) receive: enriched_signal_pack, method_executor, config
10963:
10964: 3. CANONICAL MONOLITH CONTROL:
10965:   - load_questionnaire() called ONCE by factory only (singleton + integrity hash)
10966:   - Orchestrator uses self.questionnaire object, NEVER file paths
10967:   - Search codebase: NO other load_questionnaire() calls should exist
10968:
10969: 4. SIGNAL REGISTRY CONTROL:
```

```
10970:     - create_signal_registry(questionnaire) - from canonical source ONLY
10971:     - signal_loader.py MUST BE DELETED (legacy JSON loaders eliminated)
10972:     - Registry injected into MethodExecutor, NOT accessed globally
10973:
10974: 5. ENRICHED SIGNAL PACK INJECTION:
10975:     - Factory builds EnrichedSignalPack per executor (semantic expansion + context filtering)
10976:     - Each BaseExecutor receives its specific pack, NOT full registry
10977:
10978: 6. DETERMINISM:
10979:     - SeedRegistry singleton initialized by factory for reproducibility
10980:     - ExecutorConfig encapsulates operational params (max_tokens, retries)
10981:
10982: 7. PHASE 1 HARD CONTRACTS:
10983:     - Validation constants (P01_EXPECTED_CHUNK_COUNT=60, etc.) loaded by factory
10984:     - Injected into Orchestrator for Phase 1 chunk validation
10985:     - Execution FAILS if contracts violated
10986:
10987: SIN_CARRETA Compliance:
10988:     - All construction paths emit structured telemetry with timestamps and hashes
10989:     - Determinism enforced via explicit validation of canonical questionnaire integrity
10990:     - Contract assertions guard all factory outputs (no silent degradation)
10991:     - Auditability via immutable ProcessorBundle with provenance metadata
10992: """
10993:
10994: from __future__ import annotations
10995:
10996: import hashlib
10997: import json
10998: import logging
10999: import time
11000: from collections.abc import Mapping
11001: from dataclasses import dataclass, field
11002: from typing import Any
11003:
11004: from farfan_pipeline.core.orchestrator.arg_router import ExtendedArgRouter
11005: from farfan_pipeline.core.orchestrator.class_registry import build_class_registry
11006: from farfan_pipeline.core.orchestrator.core import MethodExecutor, Orchestrator
11007: from farfan_pipeline.core.orchestrator.executor_config import ExecutorConfig
11008: from farfan_pipeline.core.orchestrator.method_registry import (
11009:     MethodRegistry,
11010:     setup_default_instantiation_rules,
11011: )
11012: from farfan_pipeline.core.orchestrator.questionnaire import (
11013:     CanonicalQuestionnaire,
11014:     load_questionnaire,
11015: )
11016: from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
11017:     EnrichedSignalPack,
11018:     create_enriched_signal_pack,
11019: )
11020: from farfan_pipeline.core.orchestrator.signal_registry import (
11021:     QuestionnaireSignalRegistry,
11022:     create_signal_registry,
11023: )
11024:
11025: # Phase 1 validation constants module
```

```
11026: try:
11027:     from farfan_pipeline.config.validation_constants import (
11028:         PHASE1_VALIDATION_CONSTANTS,
11029:         load_validation_constants,
11030:     )
11031:     VALIDATION_CONSTANTS_AVAILABLE = True
11032: except ImportError:
11033:     PHASE1_VALIDATION_CONSTANTS = {}
11034:     VALIDATION_CONSTANTS_AVAILABLE = False
11035:
11036: # Optional: CoreModuleFactory for I/O helpers
11037: try:
11038:     from farfan_pipeline.core.orchestrator.core_module_factory import CoreModuleFactory
11039:     CORE_MODULE_FACTORY_AVAILABLE = True
11040: except ImportError:
11041:     CoreModuleFactory = None # type: ignore
11042:     CORE_MODULE_FACTORY_AVAILABLE = False
11043:
11044: # SeedRegistry for determinism (REQUIRED for production)
11045: try:
11046:     from farfan_pipeline.core.orchestrator.seed_registry import SeedRegistry
11047:     SEED_REGISTRY_AVAILABLE = True
11048: except ImportError:
11049:     SeedRegistry = None # type: ignore
11050:     SEED_REGISTRY_AVAILABLE = False
11051:
11052: logger = logging.getLogger(__name__)
11053:
11054:
11055: # =====
11056: # Exceptions
11057: # =====
11058:
11059:
11060: class FactoryError(Exception):
11061:     """Base exception for factory construction failures."""
11062:     pass
11063:
11064:
11065: class QuestionnaireValidationError(FactoryError):
11066:     """Raised when questionnaire validation fails."""
11067:     pass
11068:
11069:
11070: class IntegrityError(FactoryError):
11071:     """Raised when questionnaire integrity check (SHA-256) fails."""
11072:     pass
11073:
11074:
11075: class RegistryConstructionError(FactoryError):
11076:     """Raised when signal registry construction fails."""
11077:     pass
11078:
11079:
11080: class ExecutorConstructionError(FactoryError):
11081:     """Raised when method executor construction fails."""
```

```
11082:     pass
11083:
11084:
11085: class SingletonViolationError(FactoryError):
11086:     """Raised when singleton pattern is violated."""
11087:     pass
11088:
11089:
11090: # =====
11091: # Processor Bundle (typed DI container with provenance)
11092: # =====
11093:
11094:
11095: @dataclass(frozen=True)
11096: class ProcessorBundle:
11097:     """Aggregated orchestrator dependencies built by the Factory.
11098:
11099:     This is the COMPLETE DI container returned by AnalysisPipelineFactory.
11100:
11101:     Attributes:
11102:         orchestrator: Fully configured Orchestrator (main entry point).
11103:         method_executor: MethodExecutor with signal registry injected.
11104:         questionnaire: Immutable, validated CanonicalQuestionnaire (monolith).
11105:         signal_registry: QuestionnaireSignalRegistry v2.0 from canonical source.
11106:         executor_config: ExecutorConfig for operational parameters.
11107:         enriched_signal_packs: Dict of EnrichedSignalPack per policy area.
11108:         validation_constants: Phase 1 hard contracts (chunk counts, etc.).
11109:         core_module_factory: Optional CoreModuleFactory for I/O helpers.
11110:         seed_registry_initialized: Whether SeedRegistry singleton was set up.
11111:         provenance: Construction metadata for audit trails.
11112:     """
11113:
11114:     orchestrator: Orchestrator
11115:     method_executor: MethodExecutor
11116:     questionnaire: CanonicalQuestionnaire
11117:     signal_registry: QuestionnaireSignalRegistry
11118:     executor_config: ExecutorConfig
11119:     enriched_signal_packs: dict[str, EnrichedSignalPack]
11120:     validation_constants: dict[str, Any]
11121:     core_module_factory: Any | None = None
11122:     seed_registry_initialized: bool = False
11123:     provenance: dict[str, Any] = field(default_factory=dict)
11124:
11125:     def __post_init__(self) -> None:
11126:         """SIN_CARRETA ÀS Contract Enforcement: validate bundle integrity."""
11127:         errors = []
11128:
11129:         # Critical components validation
11130:         if self.orchestrator is None:
11131:             errors.append("orchestrator must not be None")
11132:         if self.method_executor is None:
11133:             errors.append("method_executor must not be None")
11134:         if self.questionnaire is None:
11135:             errors.append("questionnaire must not be None")
11136:         if self.signal_registry is None:
11137:             errors.append("signal_registry must not be None")
```

```
11138:         if self.executor_config is None:
11139:             errors.append("executor_config must not be None")
11140:         if self.enriched_signal_packs is None:
11141:             errors.append("enriched_signal_packs must not be None")
11142:         elif not isinstance(self.enriched_signal_packs, dict):
11143:             errors.append("enriched_signal_packs must be dict[str, EnrichedSignalPack]")
11144:
11145:         if self.validation_constants is None:
11146:             errors.append("validation_constants must not be None")
11147:
11148:         # Provenance validation
11149:         if not self.provenance.get("construction_timestamp_utc"):
11150:             errors.append("provenance must include construction_timestamp_utc")
11151:         if not self.provenance.get("canonical_sha256"):
11152:             errors.append("provenance must include canonical_sha256")
11153:         if self.provenance.get("signal_registry_version") != "2.0":
11154:             errors.append("provenance must indicate signal_registry_version=2.0")
11155:
11156:         # Factory pattern enforcement check
11157:         if not self.provenance.get("factory_instantiation_confirmed"):
11158:             errors.append("provenance must confirm factory instantiation (not direct construction)")
11159:
11160:     if errors:
11161:         raise FactoryError(f"ProcessorBundle validation failed: {'; '.join(errors)}")
11162:
11163:     logger.info(
11164:         "processor_bundle_validated "
11165:         "canonical_sha256=%s construction_ts=%s policy_areas=%d validation_constants=%d",
11166:         self.provenance.get("canonical_sha256", "")[:16],
11167:         self.provenance.get("construction_timestamp_utc"),
11168:         len(self.enriched_signal_packs),
11169:         len(self.validation_constants),
11170:     )
11171:
11172:
11173: # =====
11174: # Analysis Pipeline Factory (Main Factory Class)
11175: # =====
11176:
11177:
11178: class AnalysisPipelineFactory:
11179:     """Factory for constructing the complete analysis pipeline.
11180:
11181:     This is the ONLY class that should instantiate:
11182:     - Orchestrator
11183:     - MethodExecutor
11184:     - QuestionnaireSignalRegistry
11185:     - BaseExecutor instances (30 executor classes)
11186:
11187:     CRITICAL: No other module should directly instantiate these classes.
11188:     All dependencies are injected via constructor parameters.
11189:
11190:     Usage:
11191:         factory = AnalysisPipelineFactory(
11192:             questionnaire_path="path/to/questionnaire.json",
11193:             expected_hash="abc123...",
```

```
11194:         seed=42
11195:     )
11196:     bundle = factory.create_orchestrator()
11197:     orchestrator = bundle.orchestrator
11198: """
11199:
11200: # Singleton tracking for load_questionnaire() call
11201: _questionnaire_loaded = False
11202: _questionnaire_instance: CanonicalQuestionnaire | None = None
11203:
11204: def __init__(
11205:     self,
11206:     *,
11207:     questionnaire_path: str | None = None,
11208:     expected_questionnaire_hash: str | None = None,
11209:     executor_config: ExecutorConfig | None = None,
11210:     validation_constants: dict[str, Any] | None = None,
11211:     enable_intelligence_layer: bool = True,
11212:     seed_for_determinism: int | None = None,
11213:     strict_validation: bool = True,
11214):
11215:     """Initialize the Analysis Pipeline Factory.
11216:
11217:     Args:
11218:         questionnaire_path: Path to canonical questionnaire JSON.
11219:         expected_questionnaire_hash: Expected SHA-256 hash for integrity check.
11220:         executor_config: Custom executor configuration (if None, uses default).
11221:         validation_constants: Phase 1 validation constants (if None, loads from config).
11222:         enable_intelligence_layer: Whether to build enriched signal packs.
11223:         seed_for_determinism: Seed for SeedRegistry singleton.
11224:         strict_validation: If True, fail on any validation error.
11225:
11226:         self._questionnaire_path = questionnaire_path
11227:         self._expected_hash = expected_questionnaire_hash
11228:         self._executor_config = executor_config
11229:         self._validation_constants = validation_constants
11230:         self._enable_intelligence = enable_intelligence_layer
11231:         self._seed = seed_for_determinism
11232:         self._strict = strict_validation
11233:
11234:         # Internal state (set during construction)
11235:         self._canonical_questionnaire: CanonicalQuestionnaire | None = None
11236:         self._signal_registry: QuestionnaireSignalRegistry | None = None
11237:         self._method_executor: MethodExecutor | None = None
11238:         self._enriched_packs: dict[str, EnrichedSignalPack] = {}
11239:
11240:         logger.info(
11241:             "factory_initialized questionnaire_path=%s intelligence_layer=%s seed=%s",
11242:             questionnaire_path or "default",
11243:             enable_intelligence_layer,
11244:             seed_for_determinism is not None,
11245:         )
11246:
11247:     def create_orchestrator(self) -> ProcessorBundle:
11248:         """Create fully configured Orchestrator with all dependencies injected.
11249:
```

```
11250:     This is the PRIMARY ENTRY POINT for the factory.
11251:     Returns a complete ProcessorBundle with Orchestrator ready to use.
11252:
11253:     Returns:
11254:         ProcessorBundle: Immutable bundle with all dependencies wired.
11255:
11256:     Raises:
11257:         QuestionnaireValidationError: If questionnaire validation fails.
11258:         IntegrityError: If questionnaire hash doesn't match expected.
11259:         RegistryConstructionError: If signal registry construction fails.
11260:         ExecutorConstructionError: If method executor construction fails.
11261:     """
11262:     construction_start = time.time()
11263:     timestamp_utc = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
11264:
11265:     logger.info("factory_create_orchestrator_start timestamp=%s", timestamp_utc)
11266:
11267:     try:
11268:         # Step 1: Load canonical questionnaire (ONCE, with integrity check)
11269:         self._load_canonical_questionnaire()
11270:
11271:         # Step 2: Build signal registry from canonical source
11272:         self._build_signal_registry()
11273:
11274:         # Step 3: Build enriched signal packs (intelligence layer)
11275:         self._build_enriched_signal_packs()
11276:
11277:         # Step 4: Initialize seed registry for determinism
11278:         seed_initialized = self._initialize_seed_registry()
11279:
11280:         # Step 5: Build method executor with signal registry DI
11281:         self._build_method_executor()
11282:
11283:         # Step 6: Load Phase 1 validation constants
11284:         validation_constants = self._load_validation_constants()
11285:
11286:         # Step 7: Get or create executor config
11287:         executor_config = self._get_executor_config()
11288:
11289:         # Step 8: Build orchestrator with full DI
11290:         orchestrator = self._build_orchestrator(
11291:             executor_config=executor_config,
11292:             validation_constants=validation_constants,
11293:         )
11294:
11295:         # Step 9: Assemble provenance metadata
11296:         construction_duration = time.time() - construction_start
11297:         canonical_hash = self._compute_questionnaire_hash()
11298:
11299:         provenance = {
11300:             "construction_timestamp_utc": timestamp_utc,
11301:             "canonical_sha256": canonical_hash,
11302:             "signal_registry_version": "2.0",
11303:             "intelligence_layer_enabled": self._enable_intelligence,
11304:             "enriched_packs_count": len(self._enriched_packs),
11305:             "validation_constants_count": len(validation_constants),
```

```
11306:         "construction_duration_seconds": round(construction_duration, 3),
11307:         "seed_registry_initialized": seed_initialized,
11308:         "core_module_factory_available": CORE_MODULE_FACTORY_AVAILABLE,
11309:         "strict_validation": self._strict,
11310:         "factory_instantiation_confirmed": True, # Critical for bundle validation
11311:         "factory_class": "AnalysisPipelineFactory",
11312:     }
11313:
11314:     # Step 10: Build complete bundle
11315:     bundle = ProcessorBundle(
11316:         orchestrator=orchestrator,
11317:         method_executor=self._method_executor,
11318:         questionnaire=self._canonical_questionnaire,
11319:         signal_registry=self._signal_registry,
11320:         executor_config=executor_config,
11321:         enriched_signal_packs=self._enriched_packs,
11322:         validation_constants=validation_constants,
11323:         core_module_factory=self._build_core_module_factory(),
11324:         seed_registry_initialized=seed_initialized,
11325:         provenance=provenance,
11326:     )
11327:
11328:     logger.info(
11329:         "factory_create_orchestrator_complete duration=%3fs hash=%s",
11330:         construction_duration,
11331:         canonical_hash[:16],
11332:     )
11333:
11334:     return bundle
11335:
11336: except Exception as e:
11337:     logger.error("factory_create_orchestrator_failed error=%s", str(e), exc_info=True)
11338:     raise FactoryError(f"Failed to create orchestrator: {e}") from e
11339:
11340: # =====
11341: # Internal Construction Methods
11342: # =====
11343:
11344: def _load_canonical_questionnaire(self) -> None:
11345:     """Load canonical questionnaire with singleton enforcement and integrity check.
11346:
11347:     CRITICAL REQUIREMENTS:
11348:     1. This is the ONLY place in the codebase that calls load_questionnaire()
11349:     2. Must enforce singleton pattern (only load once)
11350:     3. Must verify SHA-256 hash for integrity
11351:     4. Must raise IntegrityError if hash doesn't match
11352:
11353:     Raises:
11354:         SingletonViolationError: If load_questionnaire() already called.
11355:         IntegrityError: If questionnaire hash doesn't match expected.
11356:         QuestionnaireValidationException: If questionnaire structure invalid.
11357:     """
11358:     # Enforce singleton pattern
11359:     if AnalysisPipelineFactory._questionnaire_loaded:
11360:         if AnalysisPipelineFactory._questionnaire_instance is not None:
11361:             logger.info("questionnaire_singleton_reused using_cached_instance")
```

```
11362:             self._canonical_questionnaire = AnalysisPipelineFactory._questionnaire_instance
11363:             return
11364:         else:
11365:             raise SingletonViolationError(
11366:                 "load_questionnaire() was called but instance is None. "
11367:                 "This indicates a singleton pattern violation."
11368:             )
11369:
11370:     logger.info("questionnaire_loading_start path=%s", self._questionnaire_path or "default")
11371:
11372:     try:
11373:         # Load questionnaire (this should be the ONLY call in the entire codebase)
11374:         questionnaire = load_questionnaire(self._questionnaire_path)
11375:
11376:         # Mark singleton as loaded
11377:         AnalysisPipelineFactory._questionnaire_loaded = True
11378:         AnalysisPipelineFactory._questionnaire_instance = questionnaire
11379:
11380:         # Compute integrity hash
11381:         actual_hash = self._compute_questionnaire_hash_from_instance(questionnaire)
11382:
11383:         # Verify integrity if expected hash provided
11384:         if self._expected_hash is not None:
11385:             if actual_hash != self._expected_hash:
11386:                 raise IntegrityError(
11387:                     f"Questionnaire integrity check FAILED. "
11388:                     f"Expected: {self._expected_hash[:16]}... "
11389:                     f"Actual: {actual_hash[:16]}... "
11390:                     f"The canonical questionnaire may have been tampered with."
11391:                 )
11392:             logger.info("questionnaire_integrity_verified hash=%s", actual_hash[:16])
11393:         else:
11394:             logger.warning(
11395:                 "questionnaire_integrity_not_verified no_expected_hash_provided "
11396:                 "actual_hash=%s",
11397:                 actual_hash[:16]
11398:             )
11399:
11400:         # Validate structure
11401:         if not hasattr(questionnaire, 'questions'):
11402:             if self._strict:
11403:                 raise QuestionnaireValidationError("Questionnaire missing 'questions' attribute")
11404:             logger.warning("questionnaire_validation_warning missing_questions_attribute")
11405:
11406:         questions = getattr(questionnaire, 'questions', [])
11407:         if not questions:
11408:             if self._strict:
11409:                 raise QuestionnaireValidationError("Questionnaire has no questions")
11410:             logger.warning("questionnaire_validation_warning no_questions")
11411:
11412:         self._canonical_questionnaire = questionnaire
11413:
11414:         logger.info(
11415:             "questionnaire_loaded_successfully questions=%d hash=%s singleton=established",
11416:             len(questions),
11417:             actual_hash[:16],
```

```
11418:         )
11419:
11420:     except Exception as e:
11421:         if isinstance(e, (IntegrityError, SingletonViolationError, QuestionnaireValidationException)):
11422:             raise
11423:         raise QuestionnaireValidationException(f"Failed to load questionnaire: {e}") from e
11424:
11425:     def _build_signal_registry(self) -> None:
11426:         """Build signal registry from canonical questionnaire.
11427:
11428:             CRITICAL REQUIREMENTS:
11429:             1. Use create_signal_registry(questionnaire) ONLY
11430:             2. Pass self._canonical_questionnaire as ONLY argument
11431:             3. NO other signal loading methods allowed (signal_loader.py DELETED)
11432:
11433:             Raises:
11434:                 RegistryConstructionError: If registry construction fails.
11435:             """
11436:             if self._canonical_questionnaire is None:
11437:                 raise RegistryConstructionError(
11438:                     "Cannot build signal registry: canonical questionnaire not loaded"
11439:                 )
11440:
11441:             logger.info("signal_registry_building_start")
11442:
11443:             try:
11444:                 # Build registry from canonical source ONLY
11445:                 registry = create_signal_registry(self._canonical_questionnaire)
11446:
11447:                 # Validate registry
11448:                 if not hasattr(registry, 'get_all_policy_areas'):
11449:                     if self._strict:
11450:                         raise RegistryConstructionError("Registry missing required methods")
11451:                         logger.warning("registry_validation_warning missing_methods")
11452:
11453:                 policy_areas = registry.get_all_policy_areas() if hasattr(registry, 'get_all_policy_areas') else []
11454:
11455:                 self._signal_registry = registry
11456:
11457:                 logger.info(
11458:                     "signal_registry_built_successfully version=2.0 policy_areas=%d",
11459:                     len(policy_areas),
11460:                 )
11461:
11462:             except Exception as e:
11463:                 if isinstance(e, RegistryConstructionError):
11464:                     raise
11465:                 raise RegistryConstructionError(f"Failed to build signal registry: {e}") from e
11466:
11467:             def _build_enriched_signal_packs(self) -> None:
11468:                 """Build enriched signal packs for all policy areas.
11469:
11470:                     Each BaseExecutor receives its own EnrichedSignalPack (NOT full registry).
11471:                     Pack includes semantic expansion and context filtering.
11472:
11473:                     Raises:
```

```
11474:             RegistryConstructionError: If pack construction fails in strict mode.
11475:             """
11476:             if not self._enable_intelligence:
11477:                 logger.info("enriched_packs_disabled intelligence_layer=off")
11478:                 self._enriched_packs = {}
11479:                 return
11480:
11481:             if self._signal_registry is None:
11482:                 raise RegistryConstructionError(
11483:                     "Cannot build enriched packs: signal registry not built"
11484:                 )
11485:
11486:             logger.info("enriched_packs_building_start")
11487:
11488:             enriched_packs: dict[str, EnrichedSignalPack] = {}
11489:
11490:             try:
11491:                 policy_areas = self._signal_registry.get_all_policy_areas() if hasattr(self._signal_registry, 'get_all_policy_areas') else []
11492:
11493:                 if not policy_areas:
11494:                     logger.warning("enriched_packs_warning no_policy_areas_found")
11495:                     self._enriched_packs = enriched_packs
11496:
11497:                 return
11498:
11499:                 for policy_area_id in policy_areas:
11500:                     try:
11501:                         # Get base pack from registry
11502:                         base_pack = self._signal_registry.get(policy_area_id) if hasattr(self._signal_registry, 'get') else None
11503:
11504:                         if base_pack is None:
11505:                             logger.warning("base_pack_missing policy_area=%s", policy_area_id)
11506:                             continue
11507:
11508:                         # Create enriched pack (semantic expansion + context filtering)
11509:                         enriched_pack = create_enriched_signal_pack(
11510:                             base_pack=base_pack,
11511:                             questionnaire=self._canonical_questionnaire,
11512:                         )
11513:
11514:                         enriched_packs[policy_area_id] = enriched_pack
11515:
11516:                         logger.debug(
11517:                             "enriched_pack_created policy_area=%s",
11518:                             policy_area_id,
11519:                         )
11520:
11521:             except Exception as e:
11522:                 msg = f"Failed to create enriched pack for {policy_area_id}: {e}"
11523:                 if self._strict:
11524:                     raise RegistryConstructionError(msg) from e
11525:                     logger.error("enriched_pack_creation_failed policy_area=%s", policy_area_id, exc_info=True)
11526:
11527:             self._enriched_packs = enriched_packs
11528:
11529:             logger.info(
11530:                 "enriched_packs_built_successfully count=%d",
```

```

11530:             len(enriched_packs),
11531:         )
11532:
11533:     except Exception as e:
11534:         if isinstance(e, RegistryConstructionError):
11535:             raise
11536:         raise RegistryConstructionError(f"Failed to build enriched packs: {e}") from e
11537:
11538:     def _initialize_seed_registry(self) -> bool:
11539:         """Initialize SeedRegistry singleton for deterministic operations.
11540:
11541:         Returns:
11542:             bool: True if seed registry was initialized, False otherwise.
11543:         """
11544:         if not SEED_REGISTRY_AVAILABLE:
11545:             logger.warning("seed_registry_unavailable module_not_found determinism_not_guaranteed")
11546:             return False
11547:
11548:         if self._seed is None:
11549:             logger.info("seed_registry_not_initialized no_seed_provided")
11550:             return False
11551:
11552:         try:
11553:             SeedRegistry.initialize(master_seed=self._seed)
11554:             logger.info("seed_registry_initialized master_seed=%d determinism=enabled", self._seed)
11555:             return True
11556:         except Exception:
11557:             logger.error("seed_registry_initialization_failed", exc_info=True)
11558:             return False
11559:
11560:     def _build_method_executor(self) -> None:
11561:         """Build MethodExecutor with full dependency wiring.
11562:
11563:             CRITICAL INTEGRATION POINT - Method Dispensary Pattern:
11564:             =====
11565:
11566:             This is where the "monolith dispensaries" get wired into the pipeline.
11567:             The 30 executors orchestrate methods from these dispensaries WITHOUT
11568:             direct imports or tight coupling to the monolith implementations.
11569:
11570:             Architecture Flow:
11571:             -----
11572:             1. build_class_registry() loads the "method dispensaries" (monoliths):
11573:                 - IndustrialPolicyProcessor: 17 methods used across D1-Q1, D1-Q5, D2-Q2, D3-Q1
11574:                 - BayesianEvidenceScorer: 8 methods for confidence calculation
11575:                 - PDET MunicipalPlanAnalyzer: 52+ methods (LARGEST dispensary)
11576:                     Used in: D1-Q2, D1-Q3, D1-Q4, D2-Q1, D2-Q2, D3-Q3, D3-Q4, D3-Q5,
11577:                             D4-Q1, D4-Q2, D4-Q3, D5-Q1, D5-Q2, D5-Q4, D5-Q5
11578:                 - CausalExtractor: 28 methods for causal inference
11579:                 - FinancialAuditor: 13 methods for financial analysis
11580:                 - BayesianMechanismInference: 14 methods for mechanism testing
11581:                 - [... 15+ more classes from farfan_core]
11582:
11583:                 Total: ~240 method pairs validated (see executor_factory_validation.json)
11584:
11585:             2. These classes are NOT instantiated here - they're registered as TYPES.

```

```
11586:     Instantiation happens lazily via MethodRegistry when methods are called.
11587:
11588:     3. ExtendedArgRouter receives the class registry and provides:
11589:         - 30+ special routes for high-traffic methods (see arg_router.py)
11590:         - Generic routing via signature inspection for all other methods
11591:         - Strict argument validation (no silent parameter drops)
11592:         - **kwargs awareness for forward compatibility
11593:
11594:     4. MethodExecutor combines three critical components:
11595:         - MethodRegistry: Instantiation rules + shared instances (e.g., MunicipalOntology)
11596:         - ArgRouter: Method routing + argument validation
11597:         - SignalRegistry: Injected for signal-aware methods
11598:
11599:     5. Each of the 30 Executors orchestrates methods via:
11600:         ````python
11601:             result = self.method_executor.execute(
11602:                 class_name="PDETMunicipalPlanAnalyzer",
11603:                 method_name="_score_indicators",
11604:                 **payload # document, question_id, signal_pack, etc.
11605:             )
11606:         `````
11607:
11608:     Method Reuse Pattern:
11609:     -----
11610:     - Methods are PARTIALLY reused across executors (not fully shared)
11611:     - Example: "_score_indicators" used in D3-Q1, D3-Q2, D4-Q1
11612:     - Example: "_test_sufficiency" used in D2-Q2, D3-Q2, D3-Q4
11613:     - Each executor uses a DIFFERENT COMBINATION of methods
11614:     - Total unique combinations: 30 executors \227 avg 12 methods = ~360 method calls
11615:
11616:     Not All Methods Are Used:
11617:     -----
11618:     The monoliths contain MORE methods than executors need.
11619:     Only methods listed in executors_methods.json are actively used.
11620:     Phase 1 (ingestion) uses additional methods not in executor contracts.
11621:
11622:     Validation:
11623:     -----
11624:     - executor_factory_validation.json: 243 pairs validated, 14 failures
11625:     - Failures are methods NOT in catalog (likely private/deprecated)
11626:     - All executor contracts reference validated methods only
11627:
11628:     Signal Registry Integration:
11629:     -----
11630:     Signal registry is injected so methods can access:
11631:     - Policy-area-specific patterns
11632:     - Expected elements for validation
11633:     - Semantic enrichment via intelligence layer
11634:
11635:     Raises:
11636:         ExecutorConstructionError: If executor construction fails.
11637:
11638:     See Also:
11639:         - executors_methods.json: Complete executor\227methods mapping
11640:         - executor_factory_validation.json: Method catalog validation
11641:         - arg_router.py: Special routes and routing logic
```

```
11642:         - class_registry.py: Monolith class paths (_CLASS_PATHS)
11643:         """
11644:         if self._signal_registry is None:
11645:             raise ExecutorConstructionError(
11646:                 "Cannot build method executor: signal registry not built"
11647:             )
11648:
11649:         logger.info("method_executor_building_start dispensaries=loading")
11650:
11651:     try:
11652:         # Step 1: Build method registry with special instantiation rules
11653:         # MethodRegistry handles shared instances (e.g., MunicipalOntology singleton)
11654:         # and custom instantiation logic for complex analyzers
11655:         method_registry = MethodRegistry()
11656:         setup_default_instantiation_rules(method_registry)
11657:
11658:         logger.info("method_registry_built instantiation_rules=configured")
11659:
11660:         # Step 2: Build class registry - THE METHOD DISPENSARIES
11661:         # This loads ~20 monolith classes with 240+ methods total
11662:         # Each class is a "dispensary" that provides methods to executors
11663:         class_registry = build_class_registry()
11664:
11665:         logger.info(
11666:             "class_registry_built dispensaries=%d total_methods=240+",
11667:             len(class_registry)
11668:         )
11669:
11670:         # Step 3: Build extended arg router with special routes
11671:         # Handles 30+ high-traffic method routes + generic routing
11672:         arg_router = ExtendedArgRouter(class_registry)
11673:
11674:         special_routes = arg_router.get_special_route_coverage() if hasattr(arg_router, 'get_special_route_coverage') else 0
11675:
11676:         logger.info(
11677:             "arg_router_built special_routes=%d generic_routing=enabled",
11678:             special_routes
11679:         )
11680:
11681:         # Step 4: Build method executor WITH signal registry injected
11682:         # This is the CORE integration point - executors call methods through this
11683:         method_executor = MethodExecutor(
11684:             method_registry=method_registry,
11685:             arg_router=arg_router,
11686:             signal_registry=self._signal_registry, # DI: inject signal registry
11687:         )
11688:
11689:         # Step 5: PRE-EXECUTION CONTRACT VERIFICATION
11690:         # Verify all 30 base executor contracts (D1-Q1 through D6-Q5) before execution
11691:         # This ensures contract integrity and method class availability at startup
11692:         logger.info("contract_verification_start verifying_30_base_contracts")
11693:
11694:         from farfan_pipeline.core.orchestrator.base_executor_with_contract import (
11695:             BaseExecutorWithContract,
11696:         )
11697:
```

```
11698:         verification_result = BaseExecutorWithContract.verify_all_base_contracts(
11699:             class_registry=class_registry
11700:         )
11701:
11702:         if not verification_result["passed"]:
11703:             error_summary = f"{len(verification_result['errors'])} contract validation errors"
11704:             logger.error(
11705:                 "contract_verification_failed errors=%d warnings=%d",
11706:                 len(verification_result["errors"]),
11707:                 len(verification_result.get("warnings", [])),
11708:             )
11709:
11710:             for error in verification_result["errors"][:10]:
11711:                 logger.error("contract_error: %s", error)
11712:
11713:             if self._strict:
11714:                 raise ExecutorConstructionError(
11715:                     f"Pre-execution contract verification failed: {error_summary}. "
11716:                     f"See logs for details. Total errors: {len(verification_result['errors'])}"
11717:                 )
11718:
11719:             else:
11720:                 logger.warning(
11721:                     "contract_verification_failed_non_strict continuing_with_errors=%d",
11722:                     len(verification_result["errors"])
11723:                 )
11724:
11725:             else:
11726:                 logger.info(
11727:                     "contract_verification_passed verified=%d warnings=%d",
11728:                     len(verification_result["verified_contracts"]),
11729:                     len(verification_result.get("warnings", []))
11730:
11731:                     for warning in verification_result.get("warnings", [])[:5]:
11732:                         logger.warning("contract_warning: %s", warning)
11733:
11734: # Validate construction
11735: if not hasattr(method_executor, 'execute'):
11736:     if self._strict:
11737:         raise ExecutorConstructionError("MethodExecutor missing 'execute' method")
11738:         logger.warning("method_executor_validation_warning missing_execute")
11739:
11740:     self._method_executor = method_executor
11741:
11742:     logger.info(
11743:         "method_executor_built_successfully "
11744:         "dispensaries=%d special_routes=%d signal_registry=injected",
11745:         len(class_registry),
11746:         special_routes,
11747:     )
11748:
11749: except Exception as e:
11750:     if isinstance(e, ExecutorConstructionError):
11751:         raise
11752:         raise ExecutorConstructionError(f"Failed to build method executor: {e}") from e
11753:
11754: def _load_validation_constants(self) -> dict[str, Any]:
```

```
11754:     """Load Phase 1 validation constants (hard contracts).
11755:
11756:     These constants are injected into Orchestrator for Phase 1 validation:
11757:     - P01_EXPECTED_CHUNK_COUNT = 60
11758:     - P02_MIN_TABLE_COUNT = 5
11759:     - etc.
11760:
11761:     Returns:
11762:         dict[str, Any]: Validation constants.
11763:     """
11764:     if self._validation_constants is not None:
11765:         logger.info("validation_constants_using_provided count=%d", len(self._validation_constants))
11766:         return self._validation_constants
11767:
11768:     if VALIDATION_CONSTANTS_AVAILABLE:
11769:         try:
11770:             raw_constants = (
11771:                 load_validation_constants()
11772:                 if callable(load_validation_constants)
11773:                 else PHASE1_VALIDATION_CONSTANTS
11774:             )
11775:             if not isinstance(raw_constants, Mapping):
11776:                 raise TypeError(
11777:                     f"Validation constants must be a mapping, got {type(raw_constants)}!")
11778:             )
11779:
11780:             constants = dict(raw_constants)
11781:             logger.info("validation_constants_loaded_from_config count=%d", len(constants))
11782:             return constants
11783:         except Exception:
11784:             logger.error("validation_constants_load_failed using_defaults", exc_info=True)
11785:
11786:     # Default validation constants
11787:     default_constants = {
11788:         "P01_EXPECTED_CHUNK_COUNT": 60,
11789:         "P01_MIN_CHUNK_LENGTH": 100,
11790:         "P01_MAX_CHUNK_LENGTH": 2000,
11791:         "P02_MIN_TABLE_COUNT": 5,
11792:         "P02_MAX_TABLES_PER_DOCUMENT": 100,
11793:     }
11794:
11795:     logger.warning(
11796:         "validation_constants_using_defaults count=%d constants_module_unavailable",
11797:         len(default_constants),
11798:     )
11799:
11800:     return default_constants
11801:
11802: def _get_executor_config(self) -> ExecutorConfig:
11803:     """Get or create ExecutorConfig."""
11804:     if self._executor_config is not None:
11805:         return self._executor_config
11806:     return ExecutorConfig.default()
11807:
11808: def _build_orchestrator(
11809:     self,
```

```
11810:         executor_config: ExecutorConfig,
11811:         validation_constants: dict[str, Any],
11812:     ) -> Orchestrator:
11813:         """Build Orchestrator with full dependency injection.
11814:
11815:             CRITICAL: Orchestrator receives:
11816:             1. questionnaire: CanonicalQuestionnaire (NOT file path)
11817:             2. method_executor: MethodExecutor
11818:             3. executor_config: ExecutorConfig
11819:             4. validation_constants: dict (Phase 1 hard contracts)
11820:
11821:             Args:
11822:                 executor_config: ExecutorConfig instance.
11823:                 validation_constants: Phase 1 validation constants.
11824:
11825:             Returns:
11826:                 Orchestrator: Fully configured orchestrator.
11827:
11828:             Raises:
11829:                 ExecutorConstructionError: If orchestrator construction fails.
11830: """
11831:     if self._canonical_questionnaire is None:
11832:         raise ExecutorConstructionError("Cannot build orchestrator: questionnaire not loaded")
11833:     if self._method_executor is None:
11834:         raise ExecutorConstructionError("Cannot build orchestrator: method executor not built")
11835:
11836:     logger.info("orchestrator_building_start")
11837:
11838:     try:
11839:         # Build orchestrator with FULL dependency injection
11840:         orchestrator = Orchestrator(
11841:             questionnaire=self._canonical_questionnaire, # DI: inject questionnaire object
11842:             method_executor=self._method_executor, # DI: inject method executor
11843:             executor_config=executor_config, # DI: inject config
11844:             validation_constants=validation_constants, # DI: inject Phase 1 contracts
11845:             signal_registry=self._signal_registry, # DI: inject signal registry
11846:         )
11847:
11848:         logger.info("orchestrator_built_successfully")
11849:
11850:         return orchestrator
11851:
11852:     except Exception as e:
11853:         raise ExecutorConstructionError(f"Failed to build orchestrator: {e}") from e
11854:
11855:     def _build_core_module_factory(self) -> Any | None:
11856:         """Build CoreModuleFactory if available."""
11857:         if not CORE_MODULE_FACTORY_AVAILABLE:
11858:             return None
11859:
11860:         try:
11861:             factory = CoreModuleFactory()
11862:             logger.info("core_module_factory_built")
11863:             return factory
11864:         except Exception:
11865:             logger.error("core_module_factory_construction_error", exc_info=True)
```

```
11866:         return None
11867:
11868:     def _compute_questionnaire_hash(self) -> str:
11869:         """Compute SHA-256 hash of loaded questionnaire."""
11870:         if self._canonical_questionnaire is None:
11871:             return ""
11872:         return self._compute_questionnaire_hash_from_instance(self._canonical_questionnaire)
11873:
11874:     @staticmethod
11875:     def _compute_questionnaire_hash_from_instance(questionnaire: CanonicalQuestionnaire) -> str:
11876:         """Compute deterministic SHA-256 hash of questionnaire content."""
11877:         try:
11878:             # Try to get JSON representation if available
11879:             if hasattr(questionnaire, 'to_dict'):
11880:                 content = json.dumps(questionnaire.to_dict(), sort_keys=True)
11881:             elif hasattr(questionnaire, '__dict__'):
11882:                 content = json.dumps(questionnaire.__dict__, sort_keys=True, default=str)
11883:             else:
11884:                 content = str(questionnaire)
11885:
11886:             return hashlib.sha256(content.encode('utf-8')).hexdigest()
11887:
11888:         except Exception as e:
11889:             logger.warning("questionnaire_hash_computation_degraded error=%s", str(e))
11890:             # Fallback to simple string hash
11891:             return hashlib.sha256(str(questionnaire).encode('utf-8')).hexdigest()
11892:
11893:     def create_executor_instance(
11894:         self,
11895:         executor_class: type,
11896:         policy_area_id: str,
11897:         **extra_kwargs: Any,
11898:     ) -> Any:
11899:         """Create BaseExecutor instance with EnrichedSignalPack injected.
11900:
11901:         This method is called for each of the ~30 BaseExecutor classes.
11902:         Each executor receives its specific EnrichedSignalPack, NOT the full registry.
11903:
11904:         Args:
11905:             executor_class: BaseExecutor subclass to instantiate.
11906:             policy_area_id: Policy area identifier for signal pack selection.
11907:             **extra_kwargs: Additional kwargs to pass to constructor.
11908:
11909:         Returns:
11910:             BaseExecutor instance with dependencies injected.
11911:
11912:         Raises:
11913:             ExecutorConstructionError: If executor instantiation fails.
11914:             """
11915:             if self._method_executor is None:
11916:                 raise ExecutorConstructionError(
11917:                     "Cannot create executor: method executor not built"
11918:                 )
11919:
11920:             # Get enriched signal pack for this policy area
11921:             enriched_pack = self._enriched_packs.get(policy_area_id)
```

```
11922:         if enriched_pack is None and self._enable_intelligence:
11923:             logger.warning(
11924:                 "executor_creation_warning no_enriched_pack policy_area=%s executor=%s",
11925:                 policy_area_id,
11926:                 executor_class.__name__,
11927:             )
11928:         )
11929:
11930:     try:
11931:         # Inject dependencies into executor
11932:         executor_instance = executor_class(
11933:             method_executor=self._method_executor, # DI: inject method executor
11934:             signal_registry=self._signal_registry, # DI: inject signal registry
11935:             config=self._get_executor_config(), # DI: inject config
11936:             questionnaire_provider=self._canonical_questionnaire, # DI: inject questionnaire
11937:             enriched_pack=enriched_pack, # DI: inject enriched signal pack (specific to policy area)
11938:             **extra_kwargs,
11939:         )
11940:
11941:         logger.debug(
11942:             "executor_instance_created executor=%s policy_area=%s",
11943:             executor_class.__name__,
11944:             policy_area_id,
11945:         )
11946:
11947:         return executor_instance
11948:
11949:     except Exception as e:
11950:         raise ExecutorConstructionError(
11951:             f"Failed to create executor {executor_class.__name__}: {e}"
11952:         ) from e
11953:
11954:
11955: # =====
11956: # Convenience Functions
11957: # =====
11958:
11959:
11960: def create_analysis_pipeline(
11961:     questionnaire_path: str | None = None,
11962:     expected_hash: str | None = None,
11963:     seed: int | None = None,
11964: ) -> ProcessorBundle:
11965:     """Convenience function to create complete analysis pipeline.
11966:
11967:     This is the RECOMMENDED entry point for most use cases.
11968:
11969:     Args:
11970:         questionnaire_path: Path to canonical questionnaire JSON.
11971:         expected_hash: Expected SHA-256 hash for integrity check.
11972:         seed: Seed for reproducibility.
11973:
11974:     Returns:
11975:         ProcessorBundle with Orchestrator ready to use.
11976:     """
11977:     factory = AnalysisPipelineFactory(
```

```
11978:     questionnaire_path=questionnaire_path,
11979:     expected_questionnaire_hash=expected_hash,
11980:     seed_for_determinism=seed,
11981:     enable_intelligence_layer=True,
11982:     strict_validation=True,
11983: )
11984: return factory.create_orchestrator()
11985:
11986:
11987: def create_minimal_pipeline(
11988:     questionnaire_path: str | None = None,
11989: ) -> ProcessorBundle:
11990:     """Create minimal pipeline without intelligence layer.
11991:
11992:     Useful for testing or when enriched signals are not needed.
11993:
11994:     Args:
11995:         questionnaire_path: Path to canonical questionnaire JSON.
11996:
11997:     Returns:
11998:         ProcessorBundle with basic dependencies only.
11999: """
12000: factory = AnalysisPipelineFactory(
12001:     questionnaire_path=questionnaire_path,
12002:     enable_intelligence_layer=False,
12003:     strict_validation=False,
12004: )
12005: return factory.create_orchestrator()
12006:
12007:
12008: # =====
12009: # Validation and Diagnostics
12010: # =====
12011:
12012:
12013: def validate_factory_singleton() -> dict[str, Any]:
12014:     """Validate that load_questionnaire() was called exactly once.
12015:
12016:     Returns:
12017:         dict with validation results.
12018: """
12019:     return {
12020:         "questionnaire_loaded": AnalysisPipelineFactory._questionnaire_loaded,
12021:         "questionnaire_instance_exists": AnalysisPipelineFactory._questionnaire_instance is not None,
12022:         "singleton_pattern_valid": (
12023:             AnalysisPipelineFactory._questionnaire_loaded and
12024:             AnalysisPipelineFactory._questionnaire_instance is not None
12025:         ),
12026:     }
12027:
12028:
12029: def validate_bundle(bundle: ProcessorBundle) -> dict[str, Any]:
12030:     """Validate bundle integrity and return diagnostics."""
12031:     diagnostics = {
12032:         "valid": True,
12033:         "errors": [],
12034:     }
```

```
12034:     "warnings": [],
12035:     "components": {},
12036:     "metrics": {}
12037:   }
12038:
12039:   # Validate orchestrator
12040:   if bundle.orchestrator is None:
12041:     diagnostics["valid"] = False
12042:     diagnostics["errors"].append("orchestrator is None")
12043:   else:
12044:     diagnostics["components"]["orchestrator"] = "present"
12045:
12046:   # Validate method executor
12047:   if bundle.method_executor is None:
12048:     diagnostics["valid"] = False
12049:     diagnostics["errors"].append("method_executor is None")
12050:   else:
12051:     diagnostics["components"]["method_executor"] = "present"
12052:     if hasattr(bundle.method_executor, 'arg_router'):
12053:       router = bundle.method_executor.arg_router
12054:       if hasattr(router, 'get_special_route_coverage'):
12055:         diagnostics["metrics"]["special_routes"] = router.get_special_route_coverage()
12056:
12057:   # Validate questionnaire
12058:   if bundle.questionnaire is None:
12059:     diagnostics["valid"] = False
12060:     diagnostics["errors"].append("questionnaire is None")
12061:   else:
12062:     diagnostics["components"]["questionnaire"] = "present"
12063:     if hasattr(bundle.questionnaire, 'questions'):
12064:       diagnostics["metrics"]["question_count"] = len(bundle.questionnaire.questions)
12065:
12066:   # Validate signal registry
12067:   if bundle.signal_registry is None:
12068:     diagnostics["valid"] = False
12069:     diagnostics["errors"].append("signal_registry is None")
12070:   else:
12071:     diagnostics["components"]["signal_registry"] = "present"
12072:     if hasattr(bundle.signal_registry, 'get_all_policy_areas'):
12073:       diagnostics["metrics"]["policy_areas"] = len(bundle.signal_registry.get_all_policy_areas())
12074:
12075:   # Validate enriched packs
12076:   diagnostics["components"]["enriched_packs"] = len(bundle.enriched_signal_packs)
12077:   diagnostics["metrics"]["enriched_pack_count"] = len(bundle.enriched_signal_packs)
12078:
12079:   # Validate validation constants
12080:   diagnostics["components"]["validation_constants"] = len(bundle.validation_constants)
12081:   diagnostics["metrics"]["validation_constant_count"] = len(bundle.validation_constants)
12082:
12083:   # Validate seed registry
12084:   if not bundle.seed_registry_initialized:
12085:     diagnostics["warnings"].append("SeedRegistry not initialized - determinism not guaranteed")
12086:
12087:   # Check factory instantiation
12088:   if not bundle.provenance.get("factory_instantiation_confirmed"):
12089:     diagnostics["errors"].append("Bundle not created via AnalysisPipelineFactory")
```

```
12090:         diagnostics["valid"] = False
12091:
12092:     return diagnostics
12093:
12094:
12095: def get_bundle_info(bundle: ProcessorBundle) -> dict[str, Any]:
12096:     """Get human-readable information about bundle."""
12097:     return {
12098:         "construction_time": bundle.provenance.get("construction_timestamp_utc"),
12099:         "canonical_hash": bundle.provenance.get("canonical_sha256", "")[:16],
12100:         "policy_areas": sorted(bundle.enriched_signal_packs.keys()),
12101:         "policy_area_count": len(bundle.enriched_signal_packs),
12102:         "intelligence_layer": bundle.provenance.get("intelligence_layer_enabled"),
12103:         "validation_constants": len(bundle.validation_constants),
12104:         "construction_duration": bundle.provenance.get("construction_duration_seconds"),
12105:         "seed_initialized": bundle.seed_registry_initialized,
12106:         "factory_class": bundle.provenance.get("factory_class"),
12107:     }
12108:
12109:
12110: # =====
12111: # Module-level Checks
12112: # =====
12113:
12114:
12115: def check_legacy_signal_loader_deleted() -> dict[str, Any]:
12116:     """Check that signal_loader.py has been deleted.
12117:
12118:     Returns:
12119:         dict with check results.
12120:     """
12121:     try:
12122:         import farfan_pipeline.core.orchestrator.signal_loader
12123:         return {
12124:             "legacy_loader_deleted": False,
12125:             "error": "signal_loader.py still exists - must be deleted per architecture requirements",
12126:         }
12127:     except ImportError:
12128:         return {
12129:             "legacy_loader_deleted": True,
12130:             "message": "signal_loader.py correctly deleted - no legacy signal loading",
12131:         }
12132:
12133:
12134: def verify_single_questionnaire_load_point() -> dict[str, Any]:
12135:     """Verify that only AnalysisPipelineFactory calls load_questionnaire().
12136:
12137:     This requires manual code search but provides guidance.
12138:
12139:     Returns:
12140:         dict with verification instructions.
12141:     """
12142:     return {
12143:         "verification_required": True,
12144:         "search_command": "grep -r 'load_questionnaire(' --exclude-dir=__pycache__ --exclude='*.pyc'",
12145:         "expected_result": "Should ONLY appear in: factory.py (AnalysisPipelineFactory._load_canonical_questionnaire)",
```

```
12146:         "instructions": (
12147:             "1. Run grep command above\n"
12148:             "2. Verify ONLY factory.py calls load_questionnaire()\n"
12149:             "3. Remove any other calls found\n"
12150:             "4. Update tests to use AnalysisPipelineFactory"
12151:         ),
12152:     }
12153:
12154:
12155: def get_method_dispensary_info() -> dict[str, Any]:
12156:     """Get information about the method dispensary pattern.
12157:
12158:     Returns detailed statistics about:
12159:     - Which monolith classes serve as dispensaries
12160:     - How many methods each dispensary provides
12161:     - Which executors use which dispensaries
12162:     - Method reuse patterns
12163:
12164:     Returns:
12165:         dict with dispensary statistics and usage patterns.
12166:     """
12167:     from farfan_pipeline.core.orchestrator.class_registry import get_class_paths
12168:
12169:     class_paths = get_class_paths()
12170:
12171:     # Load executor\206\222methods mapping
12172:     try:
12173:         import json
12174:         from pathlib import Path
12175:         executors_methods_path = Path(__file__).parent / "executors_methods.json"
12176:         if executors_methods_path.exists():
12177:             with open(executors_methods_path) as f:
12178:                 executors_methods = json.load(f)
12179:         else:
12180:             executors_methods = []
12181:     except Exception:
12182:         executors_methods = []
12183:
12184:     # Build dispensary statistics
12185:     dispensaries = {}
12186:     for class_name in class_paths.keys():
12187:         dispensaries[class_name] = {
12188:             "module": class_paths[class_name],
12189:             "methods_provided": [],
12190:             "used_by_executors": [],
12191:             "total_usage_count": 0,
12192:         }
12193:
12194:     # Count method usage per dispensary
12195:     for executor_info in executors_methods:
12196:         executor_id = executor_info.get("executor_id")
12197:         methods = executor_info.get("methods", [])
12198:
12199:         for method_info in methods:
12200:             class_name = method_info.get("class")
12201:             method_name = method_info.get("method")
```

```
12202:
12203:     if class_name in dispensaries:
12204:         if method_name not in dispensaries[class_name]["methods_provided"]:
12205:             dispensaries[class_name]["methods_provided"].append(method_name)
12206:
12207:         if executor_id not in dispensaries[class_name]["used_by_executors"]:
12208:             dispensaries[class_name]["used_by_executors"].append(executor_id)
12209:
12210:             dispensaries[class_name]["total_usage_count"] += 1
12211:
12212:     # Sort by usage count
12213:     sorted_dispensaries = sorted(
12214:         dispensaries.items(),
12215:         key=lambda x: x[1]["total_usage_count"],
12216:         reverse=True
12217:     )
12218:
12219:     # Build summary statistics
12220:     total_methods = sum(len(d["methods_provided"]) for _, d in sorted_dispensaries)
12221:     total_usage = sum(d["total_usage_count"] for _, d in sorted_dispensaries)
12222:
12223:     return {
12224:         "pattern": "method_dispensary",
12225:         "description": "Monolith classes serve as method dispensaries for 30 executors",
12226:         "total_dispensaries": len(disispensaries),
12227:         "total_unique_methods": total_methods,
12228:         "total_method_calls": total_usage,
12229:         "avg_reuse_per_method": round(total_usage / max(total_methods, 1), 2),
12230:         "dispensaries": [
12231:             {
12232:                 "name": {
12233:                     "methods_count": len(info["methods_provided"]),
12234:                     "executor_count": len(info["used_by_executors"]),
12235:                     "total_calls": info["total_usage_count"],
12236:                     "reuse_factor": round(info["total_usage_count"] / max(len(info["methods_provided"]), 1), 2),
12237:                 },
12238:                 for name, info in sorted_dispensaries[:10] # Top 10
12239:             },
12240:             "top_dispensaries": [
12241:                 {
12242:                     "class": name,
12243:                     "methods": len(info["methods_provided"]),
12244:                     "executors": len(info["used_by_executors"]),
12245:                     "calls": info["total_usage_count"],
12246:                 },
12247:                 for name, info in sorted_dispensaries[:5]
12248:             ],
12249:         }
12250:
12251: def validate_method_dispensary_pattern() -> dict[str, Any]:
12252:     """Validate that the method dispensary pattern is correctly implemented.
12253:
12254:     Checks:
12255:     1. All executor methods exist in class_registry
12256:     2. No executor directly imports monolith classes
12257:     3. All methods route through MethodExecutor
```

```
12258:     4. Signal registry is injected (not globally accessed)
12259:
12260:     Returns:
12261:         dict with validation results.
12262:     """
12263:     from farfan_pipeline.core.orchestrator.class_registry import get_class_paths
12264:
12265:     class_paths = get_class_paths()
12266:     validation_results = {
12267:         "pattern_valid": True,
12268:         "errors": [],
12269:         "warnings": [],
12270:         "checks": {},
12271:     }
12272:
12273:     # Check 1: Verify class_registry is populated
12274:     if not class_paths:
12275:         validation_results["pattern_valid"] = False
12276:         validation_results["errors"].append(
12277:             "class_registry is empty - no dispensaries registered"
12278:         )
12279:     else:
12280:         validation_results["checks"]["dispensaries_registered"] = len(class_paths)
12281:
12282:     # Check 2: Verify executor_methods.json exists
12283:     try:
12284:         import json
12285:         from pathlib import Path
12286:         executors_methods_path = Path(__file__).parent / "executors_methods.json"
12287:         if not executors_methods_path.exists():
12288:             validation_results["warnings"].append(
12289:                 "executors_methods.json not found - cannot validate method mappings"
12290:             )
12291:         else:
12292:             with open(executors_methods_path) as f:
12293:                 executors_methods = json.load(f)
12294:             validation_results["checks"]["executor_method_mappings"] = len(executors_methods)
12295:     except Exception as e:
12296:         validation_results["warnings"].append(
12297:             f"Failed to load executors_methods.json: {e}"
12298:         )
12299:
12300:     # Check 3: Verify validation file exists
12301:     try:
12302:         validation_path = Path(__file__).parent / "executor_factory_validation.json"
12303:         if not validation_path.exists():
12304:             validation_results["warnings"].append(
12305:                 "executor_factory_validation.json not found - cannot validate method catalog"
12306:             )
12307:         else:
12308:             with open(validation_path) as f:
12309:                 validation_data = json.load(f)
12310:                 validation_results["checks"]["method_pairs_validated"] = validation_data.get("validated_against_catalog", 0)
12311:                 validation_results["checks"]["validation_failures"] = len(validation_data.get("failures", []))
12312:     except Exception as e:
12313:         validation_results["warnings"].append(
```

```
12314:         f"Failed to load executor_factory_validation.json: {e}"
12315:     )
12316:
12317:     return validation_results
12318:
12319:
12320:
12321: =====
12322: FILE: src/farfan_pipeline/core/orchestrator/irrigation_synchronizer.py
12323: =====
12324:
12325: """Irrigation Synchronizer - Question\206\222Chunk\206\222Task\206\222Plan Coordination.
12326:
12327: This module implements the synchronization layer that maps questionnaire questions
12328: to document chunks, generating an ExecutionPlan with 300 tasks (6 dimensions \227 50
12329: questions/dimension \227 10 policy areas) for deterministic pipeline execution.
12330:
12331: Architecture:
12332: - IrrigationSynchronizer: Orchestrates chunk\206\222question\206\222task\206\222plan flow
12333: - ExecutionPlan: Immutable plan with deterministic plan_id and integrity_hash
12334: - Task: Single unit of work (question + chunk + policy_area)
12335: - Observability: Structured JSON logs with correlation_id tracking
12336:
12337: Design Principles:
12338: - Deterministic task generation (stable ordering, reproducible plan_id)
12339: - Full observability (correlation_id propagates through all 10 phases)
12340: - Prometheus metrics for synchronization health
12341: - Blake3-based integrity hashing for plan verification
12342: """
12343:
12344: from __future__ import annotations
12345:
12346: import hashlib
12347: import json
12348: import logging
12349: import statistics
12350: import time
12351: import uuid
12352: from collections import Counter
12353: from dataclasses import dataclass, field
12354: from pathlib import Path
12355: from typing import TYPE_CHECKING, Any, Protocol
12356:
12357: if TYPE_CHECKING:
12358:     from farfan_pipeline.core.orchestrator.signals import SignalRegistry
12359:
12360:     from farfan_pipeline.core.orchestrator.task_planner import ExecutableTask
12361:     from farfan_pipeline.core.orchestrator.phase6_validation import (
12362:         validate_phase6_schema_compatibility,
12363:     )
12364:     from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
12365:     from farfan_pipeline.synchronization import ChunkMatrix
12366:
12367: try:
12368:     from farfan_pipeline.core.orchestrator.signals import (
12369:         SignalRegistry as _SignalRegistry,
```

```
12370:     )
12371: except ImportError:
12372:     _SignalRegistry = None # type: ignore
12373:
12374: try:
12375:     import blake3
12376:
12377:     BLAKE3_AVAILABLE = True
12378: except ImportError:
12379:     BLAKE3_AVAILABLE = False
12380:
12381: try:
12382:     from prometheus_client import Counter, Histogram
12383:
12384:     PROMETHEUS_AVAILABLE = True
12385: except ImportError:
12386:     PROMETHEUS_AVAILABLE = False
12387:
12388: logger = logging.getLogger(__name__)
12389:
12390: SHA256_HEX_DIGEST_LENGTH = 64
12391:
12392: SKEW_THRESHOLD_CV = 0.3
12393:
12394: class SignalRegistry(Protocol):
12395:     """Protocol for signal registry implementations.
12396:
12397:     Defines the interface that signal registries must implement for
12398:     use with IrrigationSynchronizer signal resolution.
12399:     """
12400:
12401:     def get_signals_for_chunk(
12402:         self, chunk: ChunkData, requirements: list[str]
12403:     ) -> list[Any]:
12404:         """Get signals for a chunk matching the given requirements.
12405:
12406:         Args:
12407:             chunk: Target chunk to get signals for
12408:             requirements: List of required signal types
12409:
12410:         Returns:
12411:             List of signals, each with signal_id, signal_type, and content fields
12412:         """
12413:         ...
12414:
12415:
12416: if PROMETHEUS_AVAILABLE:
12417:     synchronization_duration = Histogram(
12418:         "synchronization_duration_seconds",
12419:         "Time spent building execution plan",
12420:         buckets=[0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
12421:     )
12422:     tasks_constructed = Counter(
12423:         "synchronization_tasks_constructed_total",
12424:         "Total number of tasks constructed",
12425:         ["dimension", "policy_area"],
```

```
12426:     )
12427:     synchronization_failures = Counter(
12428:         "synchronization_failures_total",
12429:         "Total synchronization failures",
12430:         ["error_type"],
12431:     )
12432:     synchronization_chunk_matches = Counter(
12433:         "synchronization_chunk_matches_total",
12434:         "Total chunk routing matches during synchronization",
12435:         ["dimension", "policy_area", "status"],
12436:     )
12437: else:
12438:
12439:     class DummyMetric:
12440:         def time(self):
12441:             class DummyContextManager:
12442:                 def __call__(self, func):
12443:                     def wrapper(*args, **kwargs):
12444:                         return func(*args, **kwargs)
12445:
12446:                         return wrapper
12447:
12448:                     def __enter__(self):
12449:                         return self
12450:
12451:                     def __exit__(self, *args):
12452:                         pass
12453:
12454:             return DummyContextManager()
12455:
12456:         def labels(self, **kwargs):
12457:             return self
12458:
12459:         def inc(self, *args, **kwargs) -> None:
12460:             pass
12461:
12462:         synchronization_duration = DummyMetric()
12463:         tasks_constructed = DummyMetric()
12464:         synchronization_failures = DummyMetric()
12465:         synchronization_chunk_matches = DummyMetric()
12466:
12467: SHA256_HEX_DIGEST_LENGTH = 64
12468:
12469:
12470: @dataclass(frozen=True)
12471: class ChunkRoutingResult:
12472:     """Result of Phase 3 chunk routing verification.
12473:
12474:     Contains validated chunk reference and extracted metadata for task construction.
12475:
12476:
12477:     target_chunk: ChunkData
12478:     chunk_id: str
12479:     policy_area_id: str
12480:     dimension_id: str
12481:     text_content: str
```

```
12482:     expected_elements: list[dict[str, Any]]
12483:     document_position: tuple[int, int] | None
12484:
12485:
12486: @dataclass(frozen=True)
12487: class Task:
12488:     """Single unit of work in the execution plan.
12489:
12490:     Represents the mapping of one question to one chunk in a specific policy area.
12491:     """
12492:
12493:     task_id: str
12494:     dimension: str
12495:     question_id: str
12496:     policy_area: str
12497:     chunk_id: str
12498:     chunk_index: int
12499:     question_text: str
12500:
12501:
12502: @dataclass
12503: class ExecutionPlan:
12504:     """Immutable execution plan with deterministic identifiers.
12505:
12506:     Contains all tasks to be executed, with cryptographic integrity verification.
12507:     """
12508:
12509:     plan_id: str
12510:     tasks: tuple[Task, ...]
12511:     chunk_count: int
12512:     question_count: int
12513:     integrity_hash: str
12514:     created_at: str
12515:     correlation_id: str
12516:     metadata: dict[str, Any] = field(default_factory=dict)
12517:
12518:     def to_dict(self) -> dict[str, Any]:
12519:         """Convert plan to dictionary for serialization."""
12520:         return {
12521:             "plan_id": self.plan_id,
12522:             "tasks": [
12523:                 {
12524:                     "task_id": t.task_id,
12525:                     "dimension": t.dimension,
12526:                     "question_id": t.question_id,
12527:                     "policy_area": t.policy_area,
12528:                     "chunk_id": t.chunk_id,
12529:                     "chunk_index": t.chunk_index,
12530:                     "question_text": t.question_text,
12531:                 }
12532:                 for t in self.tasks
12533:             ],
12534:             "chunk_count": self.chunk_count,
12535:             "question_count": self.question_count,
12536:             "integrity_hash": self.integrity_hash,
12537:             "created_at": self.created_at,
```

```
12538:         "correlation_id": self.correlation_id,
12539:         "metadata": self.metadata,
12540:     }
12541:
12542:     @classmethod
12543:     def from_dict(cls, data: dict[str, Any]) -> ExecutionPlan:
12544:         """Reconstruct ExecutionPlan from dictionary.
12545:
12546:         Args:
12547:             data: Dictionary representation of ExecutionPlan
12548:
12549:         Returns:
12550:             ExecutionPlan instance reconstructed from dictionary
12551:         """
12552:         tasks = tuple(
12553:             Task(
12554:                 task_id=t["task_id"],
12555:                 dimension=t["dimension"],
12556:                 question_id=t["question_id"],
12557:                 policy_area=t["policy_area"],
12558:                 chunk_id=t["chunk_id"],
12559:                 chunk_index=t["chunk_index"],
12560:                 question_text=t["question_text"],
12561:             )
12562:             for t in data["tasks"]
12563:         )
12564:
12565:         return cls(
12566:             plan_id=data["plan_id"],
12567:             tasks=tasks,
12568:             chunk_count=data["chunk_count"],
12569:             question_count=data["question_count"],
12570:             integrity_hash=data["integrity_hash"],
12571:             created_at=data["created_at"],
12572:             correlation_id=data["correlation_id"],
12573:         )
12574:
12575:
12576: class IrrigationSynchronizer:
12577:     """Synchronizes questionnaire questions with document chunks.
12578:
12579:     Generates deterministic execution plans mapping questions to chunks across
12580:     all policy areas, with full observability and integrity verification.
12581:     """
12582:
12583:     def __init__(
12584:         self,
12585:         questionnaire: dict[str, Any],
12586:         preprocessed_document: PreprocessedDocument | None = None,
12587:         document_chunks: list[dict[str, Any]] | None = None,
12588:         signal_registry: SignalRegistry | None = None,
12589:     ) -> None:
12590:         """Initialize synchronizer with questionnaire and chunks.
12591:
12592:         Args:
12593:             questionnaire: Loaded questionnaire_monolith.json data
```

```
12594:     preprocessed_document: PreprocessedDocument containing validated chunks
12595:     document_chunks: Legacy list of document chunks (deprecated)
12596:     signal_registry: SignalRegistry for Phase 5 signal resolution (initialized if None)
12597:
12598:     Raises:
12599:         ValueError: If chunk matrix validation fails or no chunks provided
12600:     """
12601:     self.questionnaire = questionnaire
12602:     self.correlation_id = str(uuid.uuid4())
12603:     self.question_count = self._count_questions()
12604:     self.chunk_matrix: ChunkMatrix | None = None
12605:     self.document_chunks: list[dict[str, Any]] | None = None
12606:
12607:     if signal_registry is None and _SignalRegistry is not None:
12608:         self.signal_registry: SignalRegistry | None = _SignalRegistry()
12609:     else:
12610:         self.signal_registry = signal_registry
12611:
12612:     if preprocessed_document is not None:
12613:         try:
12614:             self.chunk_matrix = ChunkMatrix(preprocessed_document)
12615:             self.chunk_count = ChunkMatrix.EXPECTED_CHUNK_COUNT
12616:
12617:             logger.info(
12618:                 json.dumps(
12619:                     {
12620:                         "event": "irrigation_synchronizer_init",
12621:                         "correlation_id": self.correlation_id,
12622:                         "question_count": self.question_count,
12623:                         "chunk_count": self.chunk_count,
12624:                         "chunk_matrix_validated": True,
12625:                         "mode": "preprocessed_document",
12626:                         "timestamp": time.time(),
12627:                     }
12628:                 )
12629:             )
12630:         except ValueError as e:
12631:             synchronization_failures.labels(
12632:                 error_type="chunk_matrix_validation"
12633:             ).inc()
12634:             logger.error(
12635:                 json.dumps(
12636:                     {
12637:                         "event": "irrigation_synchronizer_init_failed",
12638:                         "correlation_id": self.correlation_id,
12639:                         "error": str(e),
12640:                         "error_type": "chunk_matrix_validation",
12641:                         "timestamp": time.time(),
12642:                     }
12643:                 )
12644:             )
12645:             raise ValueError(
12646:                 f"Chunk matrix validation failed during synchronizer initialization: {e}"
12647:             ) from e
12648:     elif document_chunks is not None:
12649:         self.document_chunks = document_chunks
```

```
12650:         self.chunk_count = len(document_chunks)
12651:
12652:         logger.info(
12653:             json.dumps(
12654:                 {
12655:                     "event": "irrigation_synchronizer_init",
12656:                     "correlation_id": self.correlation_id,
12657:                     "question_count": self.question_count,
12658:                     "chunk_count": self.chunk_count,
12659:                     "mode": "legacy_document_chunks",
12660:                     "timestamp": time.time(),
12661:                 }
12662:             )
12663:         )
12664:     else:
12665:         raise ValueError(
12666:             "Either preprocessed_document or document_chunks must be provided"
12667:         )
12668:
12669:     def _count_questions(self) -> int:
12670:         """Count total questions across all dimensions."""
12671:         count = 0
12672:         blocks = self.questionnaire.get("blocks", {})
12673:
12674:         for dimension_key in ["D1", "D2", "D3", "D4", "D5", "D6"]:
12675:             for i in range(1, 51):
12676:                 question_key = f"D{dimension_key[1]}_Q{i:02d}"
12677:                 if question_key in blocks:
12678:                     count += 1
12679:
12680:         return count
12681:
12682:     def validate_chunk_routing(self, question: dict[str, Any]) -> ChunkRoutingResult:
12683:         """Phase 3: Validate chunk routing and extract metadata.
12684:
12685:             Verifies that a chunk exists in the matrix for the question's routing keys,
12686:             validates chunk consistency, and extracts metadata for task construction.
12687:
12688:             Args:
12689:                 question: Question dict with routing keys (policy_area_id, dimension_id)
12690:
12691:             Returns:
12692:                 ChunkRoutingResult with validated chunk and extracted metadata
12693:
12694:             Raises:
12695:                 ValueError: If chunk not found or validation fails
12696: """
12697:         question_id = question.get("question_id", "UNKNOWN")
12698:         policy_area_id = question.get("policy_area_id")
12699:         dimension_id = question.get("dimension_id")
12700:
12701:         if not policy_area_id:
12702:             raise ValueError(
12703:                 f"Question {question_id} missing required field: policy_area_id"
12704:             )
12705:
```

```
12706:         if not dimension_id:
12707:             raise ValueError(
12708:                 f"Question {question_id} missing required field: dimension_id"
12709:             )
12710:
12711:     try:
12712:         target_chunk = self.chunk_matrix.get_chunk(policy_area_id, dimension_id)
12713:
12714:         chunk_id = target_chunk.chunk_id or f"{policy_area_id}-{dimension_id}"
12715:
12716:         if not target_chunk.text or not target_chunk.text.strip():
12717:             raise ValueError(
12718:                 f"Chunk {chunk_id} has empty text content for question {question_id}"
12719:             )
12720:
12721:         if (
12722:             target_chunk.policy_area_id
12723:             and target_chunk.policy_area_id != policy_area_id
12724:         ):
12725:             raise ValueError(
12726:                 f"Chunk routing key mismatch for {question_id}: "
12727:                 f"question policy_area={policy_area_id} but chunk has {target_chunk.policy_area_id}"
12728:             )
12729:
12730:         if target_chunk.dimension_id and target_chunk.dimension_id != dimension_id:
12731:             raise ValueError(
12732:                 f"Chunk routing key mismatch for {question_id}: "
12733:                 f"question dimension={dimension_id} but chunk has {target_chunk.dimension_id}"
12734:             )
12735:
12736:         expected_elements = question.get("expected_elements", [])
12737:
12738:         document_position = None
12739:         if target_chunk.start_pos is not None and target_chunk.end_pos is not None:
12740:             document_position = (target_chunk.start_pos, target_chunk.end_pos)
12741:
12742:         synchronization_chunk_matches.labels(
12743:             dimension=dimension_id, policy_area=policy_area_id, status="success"
12744:         ).inc()
12745:
12746:         logger.debug(
12747:             json.dumps(
12748:                 {
12749:                     "event": "chunk_routing_success",
12750:                     "question_id": question_id,
12751:                     "chunk_id": chunk_id,
12752:                     "policy_area_id": policy_area_id,
12753:                     "dimension_id": dimension_id,
12754:                     "text_length": len(target_chunk.text),
12755:                     "has_expected_elements": len(expected_elements) > 0,
12756:                     "has_document_position": document_position is not None,
12757:                     "correlation_id": self.correlation_id,
12758:                 }
12759:             )
12760:         )
12761:
```

```
12762:         return ChunkRoutingResult(
12763:             target_chunk=target_chunk,
12764:             chunk_id=chunk_id,
12765:             policy_area_id=policy_area_id,
12766:             dimension_id=dimension_id,
12767:             text_content=target_chunk.text,
12768:             expected_elements=expected_elements,
12769:             document_position=document_position,
12770:         )
12771:
12772:     except KeyError as e:
12773:         synchronization_chunk_matches.labels(
12774:             dimension=dimension_id, policy_area=policy_area_id, status="failure"
12775:         ).inc()
12776:
12777:         error_msg = (
12778:             f"Synchronization Failure for MQC {question_id}: "
12779:             f"PA={policy_area_id}, DIM={dimension_id}. "
12780:             f"No corresponding chunk found in matrix."
12781:         )
12782:
12783:         logger.error(
12784:             json.dumps(
12785:                 {
12786:                     "event": "chunk_routing_failure",
12787:                     "question_id": question_id,
12788:                     "policy_area_id": policy_area_id,
12789:                     "dimension_id": dimension_id,
12790:                     "error": error_msg,
12791:                     "correlation_id": self.correlation_id,
12792:                 }
12793:             )
12794:         )
12795:
12796:         raise ValueError(error_msg) from e
12797:
12798:     def _extract_questions(self) -> list[dict[str, Any]]:
12799:         """Extract all questions from questionnaire in deterministic order."""
12800:         questions = []
12801:         blocks = self.questionnaire.get("blocks", {})
12802:
12803:         for dimension in range(1, 7):
12804:             dim_key = f"D{dimension}"
12805:             dimension_id = f"DIM{dimension:02d}"
12806:
12807:             for q_num in range(1, 51):
12808:                 question_key = f"{dim_key}_Q{q_num:02d}"
12809:
12810:                 if question_key in blocks:
12811:                     block = blocks[question_key]
12812:                     questions.append(
12813:                         {
12814:                             "dimension": dim_key,
12815:                             "question_id": question_key,
12816:                             "question_num": q_num,
12817:                             "question_global": block.get("question_global", 0),
```

```

12818:             "question_text": block.get("question", ""),
12819:             "policy_area_id": block.get("policy_area_id"),
12820:             "dimension_id": dimension_id,
12821:             "patterns": block.get("patterns", []),
12822:             "expected_elements": block.get("expected_elements", []),
12823:             "signal_requirements": block.get("signal_requirements", {})),
12824:         }
12825:     )
12826:
12827:     questions.sort(key=lambda q: (q["dimension_id"], q["question_id"]))
12828:
12829:     return questions
12830:
12831: def _filter_patterns(
12832:     self,
12833:     patterns: list[dict[str, Any]] | tuple[dict[str, Any], ...],
12834:     policy_area_id: str,
12835: ) -> tuple[dict[str, Any], ...]:
12836:     """Filter patterns by policy_area_id using strict equality.
12837:
12838:     Filters patterns to include only those where pattern.policy_area_id == policy_area_id
12839:     (strict equality). Patterns lacking a policy_area_id attribute are excluded.
12840:
12841:     Args:
12842:         patterns: Iterable of pattern objects (typically dicts with optional policy_area_id)
12843:         policy_area_id: Policy area ID string (e.g., "PA01") to filter by
12844:
12845:     Returns:
12846:         Immutable tuple of filtered pattern dicts. Returns empty tuple if no patterns match.
12847:
12848:     Filtering Rules:
12849:         - Strict equality: pattern.policy_area_id == policy_area_id
12850:         - Exclude patterns without policy_area_id attribute
12851:         - Result is immutable (tuple)
12852:
12853:     included = []
12854:     excluded = []
12855:     included_ids = []
12856:     excluded_ids = []
12857:
12858:     for pattern in patterns:
12859:         pattern_id = (
12860:             pattern.get("id", "UNKNOWN") if isinstance(pattern, dict) else "UNKNOWN"
12861:         )
12862:
12863:         if isinstance(pattern, dict) and "policy_area_id" in pattern:
12864:             if pattern["policy_area_id"] == policy_area_id:
12865:                 included.append(pattern)
12866:                 included_ids.append(pattern_id)
12867:             else:
12868:                 excluded.append(pattern)
12869:                 excluded_ids.append(pattern_id)
12870:         else:
12871:             excluded.append(pattern)
12872:             excluded_ids.append(pattern_id)
12873:
```

```
12874:     total_count = len(included) + len(excluded)
12875:
12876:     logger.info(
12877:         json.dumps(
12878:             {
12879:                 "event": "IrrigationSynchronizer._filter_patterns",
12880:                 "total": total_count,
12881:                 "included": len(included),
12882:                 "excluded": len(excluded),
12883:                 "included_ids": included_ids,
12884:                 "excluded_ids": excluded_ids,
12885:                 "policy_area_id": policy_area_id,
12886:                 "correlation_id": self.correlation_id,
12887:             }
12888:         )
12889:     )
12890:
12891:     return tuple(included)
12892:
12893: def _construct_task(
12894:     self,
12895:     question: dict[str, Any],
12896:     routing_result: ChunkRoutingResult,
12897:     applicable_patterns: tuple[dict[str, Any], ...],
12898:     resolved_signals: tuple[Any, ...],
12899:     generated_task_ids: set[str],
12900: ) -> ExecutableTask:
12901:     """Construct ExecutableTask from question and routing result.
12902:
12903:     Extracts all fields from validated inputs, converts tuples to lists for patterns,
12904:     builds signals dict keyed by signal_type, generates creation_timestamp, populates
12905:     metadata with all required keys, validates all mandatory fields are non-None, and
12906:     catches TypeError from dataclass validation to re-raise as ValueError.
12907:
12908:     Args:
12909:         question: Question dict from questionnaire
12910:         routing_result: Validated routing result from Phase 3
12911:         applicable_patterns: Filtered tuple of patterns applicable to the routed policy area
12912:         resolved_signals: Resolved signals tuple from Phase 5
12913:         generated_task_ids: Set of task IDs generated in current synchronization run
12914:
12915:     Returns:
12916:         ExecutableTask ready for execution
12917:
12918:     Raises:
12919:         ValueError: If duplicate task_id is detected or required fields are missing/invalid
12920:     """
12921:     # Phase 7.1: Validate and extract question_global
12922:     question_global = question.get("question_global")
12923:     if question_global is None:
12924:         raise ValueError("question_global field is required but missing")
12925:     if not isinstance(question_global, int):
12926:         raise ValueError(
12927:             f"question_global must be an integer, got {type(question_global).__name__}"
12928:         )
12929:
```

```
12930:     # Phase 7.1: Construct task_id from validated question_global
12931:     task_id = f"MQC-{question_global:03d}_{routing_result.policy_area_id}"
12932:
12933:     if task_id in generated_task_ids:
12934:         raise ValueError(f"Duplicate task_id detected: {task_id}")
12935:
12936:     generated_task_ids.add(task_id)
12937:
12938:     # Field extraction in declaration order for validation priority
12939:     # Extract question_id with bracket notation and KeyError conversion
12940:     try:
12941:         question_id = question["question_id"]
12942:     except KeyError as e:
12943:         raise ValueError("question_id field is required but missing") from e
12944:
12945:     # Assign question_global (already validated above)
12946:     # Extract routing fields via attribute access (guaranteed by ChunkRoutingResult schema)
12947:     policy_area_id = routing_result.policy_area_id
12948:     dimension_id = routing_result.dimension_id
12949:     chunk_id = routing_result.chunk_id
12950:
12951:     expected_elements_list = list(routing_result.expected_elements)
12952:     document_position = routing_result.document_position
12953:
12954:     patterns_list = list(applicable_patterns)
12955:
12956:     signals_dict: dict[str, Any] = {}
12957:     for signal in resolved_signals:
12958:         if isinstance(signal, dict) and "signal_type" in signal:
12959:             signals_dict[signal["signal_type"]] = signal
12960:         elif hasattr(signal, "signal_type"):
12961:             signals_dict[signal.signal_type] = signal
12962:
12963:     from datetime import datetime, timezone
12964:
12965:     creation_timestamp = datetime.now(timezone.utc).isoformat()
12966:
12967:     metadata = {
12968:         "document_position": document_position,
12969:         "synchronizer_version": "1.0.0",
12970:         "correlation_id": self.correlation_id,
12971:         "original_pattern_count": len(applicable_patterns),
12972:         "original_signal_count": len(resolved_signals),
12973:     }
12974:
12975:     if task_id is None or not task_id:
12976:         raise ValueError("Task construction failure: task_id is None or empty")
12977:     if question_id is None or not question_id:
12978:         raise ValueError("Task construction failure: question_id is None or empty")
12979:     if question_global is None:
12980:         raise ValueError("Task construction failure: question_global is None")
12981:     if policy_area_id is None or not policy_area_id:
12982:         raise ValueError(
12983:             "Task construction failure: policy_area_id is None or empty"
12984:         )
12985:     if dimension_id is None or not dimension_id:
```

```

12986:         raise ValueError("Task construction failure: dimension_id is None or empty")
12987:     if chunk_id is None or not chunk_id:
12988:         raise ValueError("Task construction failure: chunk_id is None or empty")
12989:     if creation_timestamp is None or not creation_timestamp:
12990:         raise ValueError(
12991:             "Task construction failure: creation_timestamp is None or empty"
12992:         )
12993:
12994:     try:
12995:         task = ExecutableTask(
12996:             task_id=task_id,
12997:             question_id=question_id,
12998:             question_global=question_global,
12999:             policy_area_id=policy_area_id,
13000:             dimension_id=dimension_id,
13001:             chunk_id=chunk_id,
13002:             patterns=patterns_list,
13003:             signals=signals_dict,
13004:             creation_timestamp=creation_timestamp,
13005:             expected_elements=expected_elements_list,
13006:             metadata=metadata,
13007:         )
13008:     except TypeError as e:
13009:         raise ValueError(
13010:             f"Task construction failed for {task_id}: dataclass validation error - {e}"
13011:         ) from e
13012:
13013:     logger.debug(
13014:         f"Constructed task: task_id={task_id}, question_id={question_id}, "
13015:         f"chunk_id={chunk_id}, pattern_count={len(patterns_list)}, "
13016:         f"signal_count={len(signals_dict)}"
13017:     )
13018:
13019:     return task
13020:
13021: def _assemble_execution_plan(
13022:     self,
13023:     executable_tasks: list[ExecutableTask],
13024:     questions: list[dict[str, Any]],
13025:     correlation_id: str, # noqa: ARG002
13026: ) -> tuple[list[ExecutableTask], str]:
13027:     """Phase 8: Assemble execution plan with validation and deterministic ordering.
13028:
13029:     Performs four-phase assembly process:
13030:     - Phase 8.1: Pre-assembly validation (duplicate detection, count validation)
13031:     - Phase 8.2: Deterministic task ordering (lexicographic by task_id)
13032:     - Phase 8.3: Plan identifier computation (SHA256 of deterministic JSON)
13033:     - Phase 8.4: Plan identifier validation (format and length checks)
13034:
13035:     Validates that task count matches question count and that no duplicate
13036:     task identifiers exist. Then sorts tasks lexicographically by task_id to ensure
13037:     deterministic plan identifier generation across runs. Computes plan_id by
13038:     encoding deterministic JSON serialization (sort_keys=True, compact separators)
13039:     to UTF-8 bytes, computing SHA256 hash, and validating result matches expected
13040:     64-character lowercase hexadecimal format.
13041:
```

```
13042:     Args:
13043:         executable_tasks: List of constructed ExecutableTask objects
13044:         questions: List of question dictionaries
13045:         correlation_id: Correlation ID for tracing
13046:
13047:     Returns:
13048:         Tuple of (sorted list of ExecutableTask objects, plan_id string)
13049:
13050:     Raises:
13051:         ValueError: If task count doesn't match question count, duplicates exist,
13052:             or plan_id validation fails
13053:         RuntimeError: When sorting operation corrupts task list length
13054:
13055:     """
13056:     from collections import Counter
13057:
13058:     question_count = len(questions)
13059:     task_count = len(executable_tasks)
13060:
13061:     if task_count != question_count:
13062:         raise ValueError(
13063:             f"Execution plan assembly failure: expected {question_count} tasks "
13064:             f"but constructed {task_count}; task construction loop corrupted"
13065:         )
13066:
13067:     task_ids = [t.task_id for t in executable_tasks]
13068:     unique_count = len(set(task_ids))
13069:
13070:     if unique_count != len(task_ids):
13071:         counter = Counter(task_ids)
13072:         duplicates = [task_id for task_id, count in counter.items() if count > 1]
13073:         duplicate_count = len(task_ids) - unique_count
13074:
13075:         raise ValueError(
13076:             f"Execution plan assembly failure: found {duplicate_count} duplicate "
13077:             f"task identifiers; duplicates are {sorted(duplicates)}"
13078:         )
13079:
13080:     sorted_tasks = sorted(executable_tasks, key=lambda t: t.task_id)
13081:
13082:     if len(sorted_tasks) != len(executable_tasks):
13083:         raise RuntimeError(
13084:             f"Task ordering corruption detected: sorted task count {len(sorted_tasks)} "
13085:             f"does not match input task count {len(executable_tasks)}"
13086:
13087:     task_serialization = [
13088:         {
13089:             "task_id": t.task_id,
13090:             "question_id": t.question_id,
13091:             "question_global": t.question_global,
13092:             "policy_area_id": t.policy_area_id,
13093:             "dimension_id": t.dimension_id,
13094:             "chunk_id": t.chunk_id,
13095:         }
13096:         for t in sorted_tasks
13097:     ]
```

```

13098:
13099:     json_bytes = json.dumps(
13100:         task_serialization, sort_keys=True, separators=(", ", ":"))\
13101:         .encode("utf-8")
13102:
13103:     plan_id = hashlib.sha256(json_bytes).hexdigest()
13104:
13105:     if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
13106:         raise ValueError(
13107:             f"Plan identifier validation failure: expected length {SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
13108:             "SHA256 implementation may be compromised or monkey-patched"
13109:         )
13110:
13111:     if not all(c in "0123456789abcdef" for c in plan_id):
13112:         raise ValueError(
13113:             "Plan identifier validation failure: expected lowercase hexadecimal but got "
13114:             "characters outside '0123456789abcdef' set; SHA256 implementation may be "
13115:             "compromised or monkey-patched"
13116:         )
13117:
13118:     return sorted_tasks, plan_id
13119:
13120: def _compute_integrity_hash(self, tasks: list[Task]) -> str:
13121:     """Compute Blake3 or SHA256 integrity hash of execution plan."""
13122:     task_data = json.dumps(
13123:         [
13124:             {
13125:                 "task_id": t.task_id,
13126:                 "dimension": t.dimension,
13127:                 "question_id": t.question_id,
13128:                 "policy_area": t.policy_area,
13129:                 "chunk_id": t.chunk_id,
13130:             }
13131:             for t in tasks
13132:         ],
13133:         sort_keys=True,
13134:     ).encode("utf-8")
13135:
13136:     if BLAKE3_AVAILABLE:
13137:         return blake3.blake3(task_data).hexdigest()
13138:     else:
13139:         return hashlib.sha256(task_data).hexdigest()
13140:
13141: def _construct_execution_plan_phase_8_4(
13142:     self,
13143:     sorted_tasks: list[Task],
13144:     plan_id: str,
13145:     chunk_count: int,
13146:     question_count: int,
13147:     integrity_hash: str,
13148: ) -> ExecutionPlan:
13149:     """Phase 8.4: ExecutionPlan dataclass construction.
13150:
13151:     Constructs the final execution artifact from the sorted task list produced in
13152:     Phase 8.2, converting sorted_tasks to an immutable tuple, constructing a
13153:     metadata dictionary with generation_timestamp (UTC ISO 8601),

```

```
13154:     synchronizer_version "2.0.0", chunk_count from the chunk matrix,
13155:     question_count and task_count, invoking the ExecutionPlan constructor with
13156:     plan_id from Phase 8.3 and tasks_tuple with metadata_dict as keyword arguments,
13157:     wrapping the constructor call in try-except to catch TypeError from dataclass
13158:     validation and re-raise as ValueError with context-specific message, then
13159:     verifying task order preservation by checking that all adjacent task_id pairs
13160:     maintain lexicographic ordering and raising ValueError if any violation is
13161:     detected before emitting an info-level structured log event and returning the
13162:     constructed ExecutionPlan instance.
13163:
13164:     Args:
13165:         sorted_tasks: List of Task objects sorted by task_id (from Phase 8.2)
13166:         plan_id: Plan identifier string (from Phase 8.3)
13167:         chunk_count: Number of chunks in the document
13168:         question_count: Number of questions in the questionnaire
13169:         integrity_hash: Blake3 or SHA256 hash of the task list
13170:
13171:     Returns:
13172:         ExecutionPlan instance with validated task ordering
13173:
13174:     Raises:
13175:         ValueError: If dataclass validation fails or task ordering is violated
13176:         """
13177:     tasks_tuple = tuple(sorted_tasks)
13178:
13179:     metadata_dict = {
13180:         "generation_timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
13181:         "synchronizer_version": "2.0.0",
13182:         "chunk_count": chunk_count,
13183:         "question_count": question_count,
13184:         "task_count": len(tasks_tuple),
13185:     }
13186:
13187:     try:
13188:         plan = ExecutionPlan(
13189:             plan_id=plan_id,
13190:             tasks=tasks_tuple,
13191:             chunk_count=metadata_dict["chunk_count"],
13192:             question_count=metadata_dict["question_count"],
13193:             integrity_hash=integrity_hash,
13194:             created_at=metadata_dict["generation_timestamp"],
13195:             correlation_id=self.correlation_id,
13196:         )
13197:     except TypeError as e:
13198:         raise ValueError(
13199:             f"ExecutionPlan dataclass construction failed: {e}. "
13200:             f"Constructor validation rejected arguments (plan_id={plan_id}, "
13201:             f"task_count={len(tasks_tuple)}, chunk_count={chunk_count}, "
13202:             f"question_count={question_count})"
13203:         ) from e
13204:
13205:     for i in range(len(tasks_tuple) - 1):
13206:         current_task_id = tasks_tuple[i].task_id
13207:         next_task_id = tasks_tuple[i + 1].task_id
13208:
13209:         if current_task_id >= next_task_id:
```

```

13210:             raise ValueError(
13211:                 f"Task order preservation violation detected at index {i}: "
13212:                 f"task_id '{current_task_id}' >= task_id '{next_task_id}'. "
13213:                 f"Expected strict lexicographic ordering maintained after Phase 8.2 sort."
13214:             )
13215:
13216:     logger.info(
13217:         json.dumps(
13218:             {
13219:                 "event": "execution_plan_phase_8_4_complete",
13220:                 "plan_id": plan_id,
13221:                 "task_count": len(tasks_tuple),
13222:                 "chunk_count": chunk_count,
13223:                 "question_count": question_count,
13224:                 "integrity_hash": integrity_hash,
13225:                 "synchronizer_version": metadata_dict["synchronizer_version"],
13226:                 "generation_timestamp": metadata_dict["generation_timestamp"],
13227:                 "correlation_id": self.correlation_id,
13228:                 "phase": "execution_plan_construction_phase_8_4",
13229:             }
13230:         )
13231:     )
13232:
13233:     return plan
13234:
13235:     def _validate_cross_task_cardinality(
13236:         self, plan: ExecutionPlan, questions: list[dict[str, Any]]
13237:     ) -> None:
13238:         """Validate cross-task cardinality and log task distribution statistics.
13239:
13240:             Extracts unique chunk IDs from execution plan tasks, computes expected
13241:             reference counts by filtering questions for matching policy_area_id and
13242:             dimension_id (parsed from chunk_id), compares actual task counts per chunk
13243:             against expected counts, and emits warning-level logs for mismatches.
13244:
13245:             Also collects chunk usage statistics (mean, median, min, max) across all
13246:             unique chunks, policy area task distribution mapping, and dimension coverage
13247:             validation, culminating in a single info-level log entry with complete
13248:             observability into task distribution patterns.
13249:
13250:         Args:
13251:             plan: ExecutionPlan containing all constructed tasks
13252:             questions: List of original question dictionaries
13253:
13254:         Raises:
13255:             None - Discrepancies emit warnings but do not raise exceptions since
13256:                 they may reflect legitimate sparse coverage rather than errors
13257:         """
13258:         unique_chunks: set[str] = set()
13259:         chunk_task_counts: dict[str, int] = {}
13260:
13261:         for task in plan.tasks:
13262:             chunk_id = task.chunk_id
13263:             unique_chunks.add(chunk_id)
13264:             chunk_task_counts[chunk_id] = chunk_task_counts.get(chunk_id, 0) + 1
13265:

```

```

13266:         for chunk_id, actual_count in chunk_task_counts.items():
13267:             try:
13268:                 parts = chunk_id.split("-")
13269:                 if len(parts) >= 2:
13270:                     policy_area_id = parts[0]
13271:                     dimension_id = parts[1]
13272:
13273:                     expected_count = sum(
13274:                         1
13275:                         for q in questions
13276:                         if q.get("policy_area_id") == policy_area_id
13277:                             and q.get("dimension_id") == dimension_id
13278:                     )
13279:
13280:                     if actual_count != expected_count:
13281:                         logger.warning(
13282:                             json.dumps(
13283:                                 {
13284:                                     "event": "cross_task_cardinality_mismatch",
13285:                                     "chunk_id": chunk_id,
13286:                                     "policy_area_id": policy_area_id,
13287:                                     "dimension_id": dimension_id,
13288:                                     "expected_count": expected_count,
13289:                                     "actual_count": actual_count,
13290:                                     "correlation_id": self.correlation_id,
13291:                                     "timestamp": time.time(),
13292:                                 }
13293:                             )
13294:                         )
13295:             except (IndexError, ValueError) as e:
13296:                 logger.warning(
13297:                     json.dumps(
13298:                         {
13299:                             "event": "chunk_id_parse_error",
13300:                             "chunk_id": chunk_id,
13301:                             "error": str(e),
13302:                             "correlation_id": self.correlation_id,
13303:                             "timestamp": time.time(),
13304:                         }
13305:                     )
13306:                 )
13307:
13308:             chunk_counts = list(chunk_task_counts.values())
13309:             chunk_usage_stats: dict[str, float] = {}
13310:
13311:             if chunk_counts:
13312:                 chunk_usage_stats = {
13313:                     "mean": statistics.mean(chunk_counts),
13314:                     "median": statistics.median(chunk_counts),
13315:                     "min": float(min(chunk_counts)),
13316:                     "max": float(max(chunk_counts)),
13317:                 }
13318:
13319:             tasks_per_policy_area: dict[str, int] = {}
13320:             for task in plan.tasks:
13321:                 try:

```

```
13322:             parts = task.chunk_id.split("-")
13323:             if len(parts) >= 1:
13324:                 policy_area_id = parts[0]
13325:                 tasks_per_policy_area[policy_area_id] = (
13326:                     tasks_per_policy_area.get(policy_area_id, 0) + 1
13327:                 )
13328:             except (IndexError, ValueError):
13329:                 pass
13330:
13331:             tasks_per_dimension: dict[str, int] = {}
13332:             for task in plan.tasks:
13333:                 try:
13334:                     parts = task.chunk_id.split("-")
13335:                     if len(parts) >= 2:
13336:                         dimension_id = parts[1]
13337:                         tasks_per_dimension[dimension_id] = (
13338:                             tasks_per_dimension.get(dimension_id, 0) + 1
13339:                         )
13340:                 except (IndexError, ValueError):
13341:                     pass
13342:
13343:             logger.info(
13344:                 json.dumps(
13345:                     {
13346:                         "event": "cross_task_cardinality_validation_complete",
13347:                         "total_unique_chunks": len(unique_chunks),
13348:                         "tasks_per_policy_area": tasks_per_policy_area,
13349:                         "tasks_per_dimension": tasks_per_dimension,
13350:                         "chunk_usage_stats": chunk_usage_stats,
13351:                         "correlation_id": self.correlation_id,
13352:                         "timestamp": time.time(),
13353:                     }
13354:                 )
13355:             )
13356:
13357:             @synchronization_duration.time()
13358:             def build_execution_plan(self) -> ExecutionPlan:
13359:                 """Build deterministic execution plan mapping questions to chunks.
13360:
13361:                 Uses validated chunk matrix if available, otherwise falls back to
13362:                 legacy document_chunks iteration mode.
13363:
13364:                 Returns:
13365:                     ExecutionPlan with deterministic plan_id and integrity_hash
13366:
13367:                 Raises:
13368:                     ValueError: If question data is invalid or chunk matrix lookup fails
13369:                     """
13370:                 if self.chunk_matrix is not None:
13371:                     return self._build_with_chunk_matrix()
13372:                 else:
13373:                     return self._build_with_legacy_chunks()
13374:
13375:             def _build_with_chunk_matrix(self) -> ExecutionPlan:
13376:                 """Build execution plan using validated chunk matrix.
13377:
```

```
13378:     Orchestrates Phases 2-7 of irrigation synchronization:
13379:     - Phase 2: Question extraction
13380:     - Phase 3: Chunk routing (OBJECTIVE 3 INTEGRATION)
13381:     - Phase 4: Pattern filtering (policy_area_id-based filtering)
13382:     - Phase 5: Signal resolution (future)
13383:     - Phase 6: Schema validation (future)
13384:     - Phase 7: Task construction
13385:
13386:     Returns:
13387:         ExecutionPlan with validated tasks
13388:
13389:     Raises:
13390:         ValueError: On routing failures, validation errors
13391:         """
13392:         logger.info(
13393:             json.dumps(
13394:                 {
13395:                     "event": "task_construction_start",
13396:                     "correlation_id": self.correlation_id,
13397:                     "question_count": self.question_count,
13398:                     "chunk_count": self.chunk_count,
13399:                     "mode": "chunk_matrix",
13400:                     "phase": "synchronization_phase_2",
13401:                     "timestamp": time.time(),
13402:                 }
13403:             )
13404:         )
13405:
13406:     try:
13407:         if self.question_count == 0:
13408:             synchronization_failures.labels(error_type="empty_questions").inc()
13409:             raise ValueError(
13410:                 "No questions extracted from questionnaire. "
13411:                 "Cannot build tasks with empty question set."
13412:             )
13413:
13414:         questions = self._extract_questions()
13415:
13416:         if not questions:
13417:             raise ValueError(
13418:                 "No questions extracted from questionnaire. "
13419:                 "Cannot build tasks with empty question set."
13420:             )
13421:
13422:         tasks: list[ExecutableTask] = []
13423:         routing_successes = 0
13424:         routing_failures = 0
13425:         generated_task_ids: set[str] = set()
13426:
13427:         for idx, question in enumerate(questions, start=1):
13428:             question_id = question.get("question_id", f"UNKNOWN_{idx}")
13429:             policy_area_id = question.get("policy_area_id", "UNKNOWN")
13430:             dimension_id = question.get("dimension_id", "UNKNOWN")
13431:             chunk_id = "UNKNOWN"
13432:
13433:             try:
```

```
13434:             routing_result = self.validate_chunk_routing(question)
13435:             routing_successes += 1
13436:             chunk_id = routing_result.chunk_id
13437:
13438:             patterns_raw = question.get("patterns", [])
13439:             applicable_patterns = self._filter_patterns(
13440:                 patterns_raw, routing_result.policy_area_id
13441:             )
13442:
13443:             # Phase 5 validation: Ensure signal_registry initialized
13444:             if self.signal_registry is None:
13445:                 raise ValueError(
13446:                     f"SignalRegistry required for Phase 5 signal resolution "
13447:                     f"but not initialized for question {question_id}"
13448:                 )
13449:
13450:             resolved_signals = self._resolve_signals_for_question(
13451:                 question,
13452:                 routing_result.target_chunk,
13453:                 self.signal_registry,
13454:             )
13455:
13456:             # Phase 6: Schema validation (four subphase pipeline)
13457:             # Validates structural compatibility and semantic constraints
13458:             # Allows TypeError/ValueError to propagate to outer handler
13459:             validate_phase6_schema_compatibility(
13460:                 question=question,
13461:                 chunk_expected_elements=routing_result.expected_elements,
13462:                 chunk_id=routing_result.chunk_id,
13463:                 policy_area_id=routing_result.policy_area_id,
13464:                 correlation_id=self.correlation_id,
13465:             )
13466:
13467:             task = self._construct_task(
13468:                 question,
13469:                 routing_result,
13470:                 applicable_patterns,
13471:                 resolved_signals,
13472:                 generated_task_ids,
13473:             )
13474:             tasks.append(task)
13475:
13476:             if idx % 50 == 0:
13477:                 logger.info(
13478:                     json.dumps(
13479:                         {
13480:                             "event": "task_construction_progress",
13481:                             "tasks_completed": idx,
13482:                             "total_questions": len(questions),
13483:                             "progress_pct": round(
13484:                                 100 * idx / len(questions), 2
13485:                             ),
13486:                             "correlation_id": self.correlation_id,
13487:                         }
13488:                     )
13489:                 )
```

```
13490:
13491:             except (ValueError, TypeError) as e:
13492:                 routing_failures += 1
13493:
13494:                 logger.error(
13495:                     json.dumps(
13496:                         {
13497:                             "event": "task_construction_failure",
13498:                             "error_event": "routing_or_signal_failure",
13499:                             "question_id": question_id,
13500:                             "question_index": idx,
13501:                             "policy_area_id": policy_area_id,
13502:                             "dimension_id": dimension_id,
13503:                             "chunk_id": chunk_id,
13504:                             "error_type": type(e).__name__,
13505:                             "error_message": str(e),
13506:                             "correlation_id": self.correlation_id,
13507:                             "timestamp": time.time(),
13508:                         }
13509:                     ),
13510:                     exc_info=True,
13511:                 )
13512:
13513:             raise
13514:
13515:     expected_task_count = len(questions)
13516:     actual_task_count = len(tasks)
13517:
13518:     if actual_task_count != expected_task_count:
13519:         raise ValueError(
13520:             f"Task count mismatch: Expected {expected_task_count} tasks "
13521:             f"but constructed {actual_task_count}. "
13522:             f"Routing successes: {routing_successes}, failures: {routing_failures}"
13523:         )
13524:
13525:     tasks, plan_id = self._assemble_execution_plan(
13526:         tasks, questions, self.correlation_id
13527:     )
13528:
13529:     logger.info(
13530:         json.dumps(
13531:             {
13532:                 "event": "task_construction_complete",
13533:                 "total_tasks": actual_task_count,
13534:                 "routing_successes": routing_successes,
13535:                 "routing_failures": routing_failures,
13536:                 "success_rate": round(
13537:                     100 * routing_successes / max(expected_task_count, 1), 2
13538:                 ),
13539:                 "correlation_id": self.correlation_id,
13540:                 "timestamp": time.time(),
13541:             }
13542:         )
13543:     )
13544:
13545:     legacy_tasks = []
```

```
13546:         for task in tasks:
13547:             legacy_task = Task(
13548:                 task_id=task.task_id,
13549:                 dimension=task.dimension_id,
13550:                 question_id=task.question_id,
13551:                 policy_area=task.policy_area_id,
13552:                 chunk_id=task.chunk_id,
13553:                 chunk_index=0,
13554:                 question_text="",
13555:             )
13556:             legacy_tasks.append(legacy_task)
13557:
13558:     integrity_hash = self._compute_integrity_hash(legacy_tasks)
13559:
13560:     plan = self._construct_execution_plan_phase_8_4(
13561:         sorted_tasks=legacy_tasks,
13562:         plan_id=plan_id,
13563:         chunk_count=self.chunk_count,
13564:         question_count=len(questions),
13565:         integrity_hash=integrity_hash,
13566:     )
13567:
13568:     self._validate_cross_task_cardinality(plan, questions)
13569:
13570:     logger.info(
13571:         json.dumps(
13572:             {
13573:                 "event": "build_execution_plan_complete",
13574:                 "correlation_id": self.correlation_id,
13575:                 "plan_id": plan_id,
13576:                 "task_count": len(legacy_tasks),
13577:                 "chunk_count": self.chunk_count,
13578:                 "question_count": len(questions),
13579:                 "integrity_hash": integrity_hash,
13580:                 "chunk_matrix_validated": True,
13581:                 "mode": "chunk_matrix",
13582:                 "phase": "synchronization_phase_complete",
13583:             }
13584:         )
13585:     )
13586:
13587:     return plan
13588:
13589: except ValueError as e:
13590:     synchronization_failures.labels(error_type="validation_failure").inc()
13591:     logger.error(
13592:         json.dumps(
13593:             {
13594:                 "event": "build_execution_plan_error",
13595:                 "correlation_id": self.correlation_id,
13596:                 "error": str(e),
13597:                 "error_type": "validation_failure",
13598:             }
13599:         )
13600:     )
13601:     raise
```

```
13602:         except Exception as e:
13603:             synchronization_failures.labels(error_type=type(e).__name__).inc()
13604:             logger.error(
13605:                 json.dumps(
13606:                     {
13607:                         "event": "build_execution_plan_error",
13608:                         "correlation_id": self.correlation_id,
13609:                         "error": str(e),
13610:                         "error_type": type(e).__name__,
13611:                     }
13612:                 )
13613:             )
13614:             raise
13615:
13616:     def _build_with_legacy_chunks(self) -> ExecutionPlan:
13617:         """Build execution plan using legacy document_chunks list."""
13618:         logger.info(
13619:             json.dumps(
13620:                 {
13621:                     "event": "build_execution_plan_start",
13622:                     "correlation_id": self.correlation_id,
13623:                     "question_count": self.question_count,
13624:                     "chunk_count": self.chunk_count,
13625:                     "mode": "legacy_chunks",
13626:                     "phase": "synchronization_phase_0",
13627:                 }
13628:             )
13629:         )
13630:
13631:         try:
13632:             if not self.document_chunks:
13633:                 synchronization_failures.labels(error_type="empty_chunks").inc()
13634:                 raise ValueError("No document chunks provided")
13635:
13636:             if self.question_count == 0:
13637:                 synchronization_failures.labels(error_type="empty_questions").inc()
13638:                 raise ValueError("No questions found in questionnaire")
13639:
13640:             questions = self._extract_questions()
13641:             policy_areas = [f"PA{i:02d}" for i in range(1, 11)]
13642:
13643:             tasks: list[Task] = []
13644:
13645:             for question in questions:
13646:                 for policy_area in policy_areas:
13647:                     for chunk_idx, chunk in enumerate(self.document_chunks):
13648:                         chunk_id = chunk.get("chunk_id", f"chunk_{chunk_idx:04d}")
13649:
13650:                         task_id = f"{question['question_id']}_{policy_area}_{chunk_id}"
13651:
13652:                         task = Task(
13653:                             task_id=task_id,
13654:                             dimension=question["dimension"],
13655:                             question_id=question["question_id"],
13656:                             policy_area=policy_area,
13657:                             chunk_id=chunk_id,
```

```
13658:                 chunk_index=chunk_idx,
13659:                 question_text=question["question_text"],
13660:             )
13661:             tasks.append(task)
13662:         tasks_constructed.labels(
13663:             dimension=question["dimension"], policy_area=policy_area
13664:         ).inc()
13665:     )
13666:     sorted_tasks = sorted(tasks, key=lambda t: t.task_id)
13667: 
13668:     if len(sorted_tasks) != len(tasks):
13669:         raise RuntimeError(
13670:             f"Task ordering corruption detected: sorted task count {len(sorted_tasks)} "
13671:             f"does not match input task count {len(tasks)}"
13672:         )
13673: 
13674:         task_serialization = [
13675:             {
13676:                 "task_id": t.task_id,
13677:                 "question_id": t.question_id,
13678:                 "dimension": t.dimension,
13679:                 "policy_area": t.policy_area,
13680:                 "chunk_id": t.chunk_id,
13681:             }
13682:             for t in sorted_tasks
13683:         ]
13684: 
13685:     json_bytes = json.dumps(
13686:         task_serialization, sort_keys=True, separators=(", ", ":")

13687:     ).encode("utf-8")
13688: 
13689:     plan_id = hashlib.sha256(json_bytes).hexdigest()
13690: 
13691:     if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
13692:         raise ValueError(
13693:             f"Plan identifier validation failure: expected length {SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
13694:             "SHA256 implementation may be compromised or monkey-patched"
13695:         )
13696: 
13697:         )
13698: 
13699:     if not all(c in "0123456789abcdef" for c in plan_id):
13700:         raise ValueError(
13701:             "Plan identifier validation failure: expected lowercase hexadecimal but got "
13702:             "characters outside '0123456789abcdef' set; SHA256 implementation may be "
13703:             "compromised or monkey-patched"
13704:         )
13705: 
13706:     integrity_hash = self._compute_integrity_hash(sorted_tasks)
13707: 
13708:     plan = self._construct_execution_plan_phase_8_4(
13709:         sorted_tasks=sorted_tasks,
13710:         plan_id=plan_id,
13711:         chunk_count=self.chunk_count,
13712:         question_count=len(questions),
13713:         integrity_hash=integrity_hash,
```

```
13714:         )
13715:
13716:         self._validate_cross_task_cardinality(plan, questions)
13717:
13718:         logger.info(
13719:             json.dumps(
13720:                 {
13721:                     "event": "build_execution_plan_complete",
13722:                     "correlation_id": self.correlation_id,
13723:                     "plan_id": plan_id,
13724:                     "task_count": len(tasks),
13725:                     "chunk_count": self.chunk_count,
13726:                     "question_count": len(questions),
13727:                     "integrity_hash": integrity_hash,
13728:                     "mode": "legacy_chunks",
13729:                     "phase": "synchronization_phase_complete",
13730:                 }
13731:             )
13732:         )
13733:
13734:         return plan
13735:
13736:     except Exception as e:
13737:         synchronization_failures.labels(error_type=type(e).__name__).inc()
13738:         logger.error(
13739:             json.dumps(
13740:                 {
13741:                     "event": "build_execution_plan_error",
13742:                     "correlation_id": self.correlation_id,
13743:                     "error": str(e),
13744:                     "error_type": type(e).__name__,
13745:                 }
13746:             )
13747:         )
13748:         raise
13749:
13750:     def _validate_cross_task_contamination(self, execution_plan: ExecutionPlan) -> None:
13751:         """Build traceability mappings for task-chunk relationship queries.
13752:
13753:             Constructs two bidirectional dictionaries enabling efficient task-chunk
13754:             relationship queries and stores them in ExecutionPlan metadata:
13755:             - task_chunk_mapping: Maps each task_id to its chunk_id (one-to-one)
13756:             - chunk_task_mapping: Maps each chunk_id to list of task_ids (one-to-many)
13757:
13758:             Args:
13759:                 execution_plan: ExecutionPlan to enrich with traceability mappings
13760:
13761:             Returns:
13762:                 None (modifies execution_plan.metadata in place)
13763: """
13764:     task_chunk_mapping = {t.task_id: t.chunk_id for t in execution_plan.tasks}
13765:
13766:     chunk_task_mapping: dict[str, list[str]] = {}
13767:     for t in execution_plan.tasks:
13768:         chunk_task_mapping.setdefault(t.chunk_id, []).append(t.task_id)
13769:
```

```
13770:     execution_plan.metadata["task_chunk_mapping"] = task_chunk_mapping
13771:     execution_plan.metadata["chunk_task_mapping"] = chunk_task_mapping
13772:
13773:     def _resolve_signals_for_question(
13774:         self,
13775:         question: dict[str, Any],
13776:         target_chunk: ChunkData,
13777:         signal_registry: SignalRegistry,
13778:     ) -> tuple[Any, ...]:
13779:         """Resolve signals for a question from registry.
13780:
13781:             Performs signal resolution with comprehensive validation:
13782:             - Normalizes signal_requirements to empty list if missing/None
13783:             - Calls signal_registry.get_signals_for_chunk with requirements
13784:             - Validates return type is list (raises TypeError if None)
13785:             - Validates each signal has required fields (signal_id, signal_type, content)
13786:             - Detects missing required signals (HARD STOP with ValueError)
13787:             - Detects and warns about duplicate signal types
13788:             - Returns immutable tuple of resolved signals
13789:
13790:         Args:
13791:             question: Question dict with signal_requirements field
13792:             target_chunk: Target ChunkData for signal resolution
13793:             signal_registry: Registry implementing get_signals_for_chunk(chunk, requirements)
13794:
13795:         Returns:
13796:             Immutable tuple of resolved signals
13797:
13798:         Raises:
13799:             TypeError: If signal_registry returns non-list type
13800:             ValueError: If signal missing required field or required signals not found
13801: """
13802:     question_id = question.get("question_id", "UNKNOWN")
13803:     chunk_id = getattr(target_chunk, "chunk_id", "UNKNOWN")
13804:
13805:     # Normalize signal_requirements to empty list if missing or None
13806:     signal_requirements = question.get("signal_requirements")
13807:     if signal_requirements is None:
13808:         signal_requirements = []
13809:     elif not isinstance(signal_requirements, list):
13810:         # If it's a dict or other type, extract as list if possible
13811:         if isinstance(signal_requirements, dict):
13812:             signal_requirements = list(signal_requirements.keys())
13813:         else:
13814:             signal_requirements = []
13815:
13816:     # Call signal_registry.get_signals_for_chunk
13817:     resolved_signals = signal_registry.get_signals_for_chunk(
13818:         target_chunk, signal_requirements
13819:     )
13820:
13821:     # Validate return is list type (raise TypeError if None)
13822:     if resolved_signals is None:
13823:         raise TypeError(
13824:             f"SignalRegistry returned {type(None).__name__} for question {question_id} "
13825:             f"chunk {chunk_id}, expected list"
```

```
13826:         )
13827:
13828:     if not isinstance(resolved_signals, list):
13829:         raise TypeError(
13830:             f"SignalRegistry returned {type(resolved_signals).__name__} for question {question_id} "
13831:             f"chunk {chunk_id}, expected list"
13832:         )
13833:
13834:     # Validate each signal has required fields
13835:     required_fields = ["signal_id", "signal_type", "content"]
13836:     for i, signal in enumerate(resolved_signals):
13837:         for field in required_fields:
13838:             # Try both attribute and dict access
13839:             has_field = False
13840:             try:
13841:                 if hasattr(signal, field):
13842:                     setattr(signal, field)
13843:                     has_field = True
13844:             except (AttributeError, KeyError):
13845:                 pass
13846:
13847:             if not has_field:
13848:                 try:
13849:                     if isinstance(signal, dict) and field in signal:
13850:                         has_field = True
13851:                     except (TypeError, KeyError):
13852:                         pass
13853:
13854:                 if not has_field:
13855:                     raise ValueError(
13856:                         f"Signal at index {i} missing field {field} for question {question_id}"
13857:                     )
13858:
13859:     # Extract signal_types into set
13860:     signal_types = set()
13861:     for signal in resolved_signals:
13862:         # Try attribute access first, then dict access
13863:         signal_type = None
13864:         try:
13865:             if hasattr(signal, "signal_type"):
13866:                 signal_type = signal.signal_type
13867:             except AttributeError:
13868:                 pass
13869:
13870:             if signal_type is None:
13871:                 try:
13872:                     if isinstance(signal, dict):
13873:                         signal_type = signal["signal_type"]
13874:                     except (KeyError, TypeError):
13875:                         pass
13876:
13877:             if signal_type is not None:
13878:                 signal_types.add(signal_type)
13879:
13880:     # Compute missing signals
13881:     requirements_set = set(signal_requirements) if signal_requirements else set()
```

```
13882:     missing_signals = requirements_set - signal_types
13883:
13884:     # Raise ValueError if non-empty (HARD STOP)
13885:     if missing_signals:
13886:         missing_sorted = sorted(missing_signals)
13887:         raise ValueError(
13888:             f"Synchronization Failure for MOC {question_id}: "
13889:             f"Missing required signals {missing_sorted} for chunk {chunk_id}"
13890:         )
13891:
13892:     # Detect duplicates
13893:     if len(resolved_signals) > len(signal_types):
13894:         # Find duplicate types for logging
13895:         type_counts: dict[Any, int] = {}
13896:         for signal in resolved_signals:
13897:             signal_type = None
13898:             try:
13899:                 if hasattr(signal, "signal_type"):
13900:                     signal_type = signal.signal_type
13901:             except AttributeError:
13902:                 pass
13903:
13904:             if signal_type is None:
13905:                 try:
13906:                     if isinstance(signal, dict):
13907:                         signal_type = signal["signal_type"]
13908:                 except (KeyError, TypeError):
13909:                     pass
13910:
13911:             if signal_type is not None:
13912:                 type_counts[signal_type] = type_counts.get(signal_type, 0) + 1
13913:
13914:     duplicate_types = [t for t, count in type_counts.items() if count > 1]
13915:
13916:     logger.warning(
13917:         "signal_resolution_duplicates",
13918:         extra={
13919:             "question_id": question_id,
13920:             "chunk_id": chunk_id,
13921:             "correlation_id": self.correlation_id,
13922:             "duplicate_types": duplicate_types,
13923:         },
13924:     )
13925:
13926:     # Emit success log
13927:     logger.debug(
13928:         "signal_resolution_success",
13929:         extra={
13930:             "question_id": question_id,
13931:             "chunk_id": chunk_id,
13932:             "correlation_id": self.correlation_id,
13933:             "resolved_count": len(resolved_signals),
13934:             "required_count": len(signal_requirements),
13935:             "signal_types": list(signal_types),
13936:         },
13937:     )
```

```
13938:
13939:     # Return tuple for immutability
13940:     return tuple(resolved_signals)
13941:
13942:     def _serialize_and_verify_plan(self, plan: ExecutionPlan) -> str:
13943:         """Serialize ExecutionPlan and verify round-trip integrity.
13944:
13945:             Serializes the execution plan to JSON, deserializes it back, reconstructs
13946:             an ExecutionPlan instance, and validates that plan_id and task count match
13947:             the original to ensure serialization is lossless.
13948:
13949:             Args:
13950:                 plan: ExecutionPlan instance to serialize and verify
13951:
13952:             Returns:
13953:                 Validated serialized JSON string ready for persistent storage
13954:
13955:             Raises:
13956:                 ValueError: If plan_id mismatch or task count mismatch detected
13957: """
13958:     plan_dict = plan.to_dict()
13959:     serialized_json = json.dumps(plan_dict, sort_keys=True, separators=(", ", " :"))
13960:
13961:     deserialized_dict = json.loads(serialized_json)
13962:     reconstructed_plan = ExecutionPlan.from_dict(deserialized_dict)
13963:
13964:     if reconstructed_plan.plan_id != plan.plan_id:
13965:         raise ValueError(
13966:             f"Serialization verification failed: plan_id mismatch "
13967:             f"(original={plan.plan_id}, reconstructed={reconstructed_plan.plan_id})"
13968:         )
13969:
13970:     original_task_count = len(plan.tasks)
13971:     reconstructed_task_count = len(reconstructed_plan.tasks)
13972:
13973:     if reconstructed_task_count != original_task_count:
13974:         raise ValueError(
13975:             f"Serialization verification failed: task count mismatch "
13976:             f"(original={original_task_count}, reconstructed={reconstructed_task_count})"
13977:         )
13978:
13979:     return serialized_json
13980:
13981:     def _archive_to_storage(
13982:         self,
13983:         serialized_json: str,
13984:         execution_plan: ExecutionPlan,
13985:         base_dir: Path,
13986:     ) -> ExecutionPlan:
13987:         """Archive execution plan to storage with atomic index update and rollback.
13988:
13989:             Constructs storage path as base_dir / 'execution_plans' / f'{plan_id}.json',
13990:             writes serialized JSON with verification, and atomically updates index with
13991:             rollback logic for orphaned files.
13992:
13993:             Args:
```

```
13994:     serialized_json: Serialized JSON string of execution plan
13995:     execution_plan: ExecutionPlan instance to archive
13996:     base_dir: Base directory path for storage
13997:
13998:     Returns:
13999:         Original ExecutionPlan instance unchanged
14000:
14001:     Raises:
14002:         ValueError: If write fails (re-raised from IOError)
14003:         IOError: If write verification fails (content mismatch)
14004:
14005:     """
14006:     plan_id = execution_plan.plan_id
14007:     storage_path = base_dir / "execution_plans" / f"{plan_id}.json"
14008:
14009:     try:
14010:         storage_path.parent.mkdir(parents=True, exist_ok=True)
14011:     except IOError as e:
14012:         raise ValueError(
14013:             f"Failed to create parent directories for plan_id={plan_id}, "
14014:             f"storage_path={storage_path}: {e}"
14015:         ) from e
14016:
14017:     try:
14018:         storage_path.write_text(serialized_json, encoding="utf-8")
14019:     except IOError as e:
14020:         raise ValueError(
14021:             f"Failed to write execution plan for plan_id={plan_id}, "
14022:             f"storage_path={storage_path}: {e}"
14023:         ) from e
14024:
14025:     try:
14026:         read_content = storage_path.read_text(encoding="utf-8")
14027:         if read_content != serialized_json:
14028:             storage_path.unlink()
14029:             raise IOError(
14030:                 f"Write verification failed for plan_id={plan_id}, "
14031:                 f"storage_path={storage_path}: content mismatch after write"
14032:             )
14033:     except IOError as e:
14034:         if storage_path.exists():
14035:             storage_path.unlink()
14036:
14037:     index_path = base_dir / "execution_plans" / "index.jsonl"
14038:     index_entry = {
14039:         "plan_id": plan_id,
14040:         "storage_path": str(storage_path),
14041:         "created_at": execution_plan.created_at,
14042:         "task_count": len(execution_plan.tasks),
14043:         "integrity_hash": execution_plan.integrity_hash,
14044:         "correlation_id": execution_plan.correlation_id,
14045:     }
14046:
14047:     try:
14048:         with open(index_path, "a", encoding="utf-8") as f:
14049:             f.write(json.dumps(index_entry) + "\n")
```

```
14050:         except IOError as e:
14051:             if storage_path.exists():
14052:                 storage_path.unlink()
14053:             raise ValueError(
14054:                 f"Failed to update index for plan_id={plan_id}, "
14055:                 f"storage_path={storage_path}: {e}"
14056:             ) from e
14057:
14058:     logger.info(
14059:         "execution_plan_archived",
14060:         extra={
14061:             "event": "execution_plan_archived",
14062:             "plan_id": plan_id,
14063:             "storage_path": str(storage_path),
14064:             "task_count": len(execution_plan.tasks),
14065:             "integrity_hash": execution_plan.integrity_hash,
14066:             "correlation_id": execution_plan.correlation_id,
14067:             "created_at": execution_plan.created_at,
14068:         },
14069:     )
14070:
14071:     return execution_plan
14072:
14073:
14074: __all__ = [
14075:     "IrrigationSynchronizer",
14076:     "ExecutionPlan",
14077:     "Task",
14078:     "ChunkRoutingResult",
14079:     "SignalRegistry",
14080: ]
14081:
14082:
14083:
14084: =====
14085: FILE: src/farfan_pipeline/core/orchestrator/memory_safety.py
14086: =====
14087:
14088: """
14089: Memory Safety Guards for Executor System
14090:
14091: Provides systematic memory safety across all executors processing large objects
14092: (entities, DAGs, causal effects, etc.) with:
14093: - Size estimation for Python objects and JSON serialization
14094: - Configurable limits per executor type
14095: - Memory pressure detection using psutil
14096: - Fallback strategies (sampling, truncation) with logging and metrics
14097: """
14098:
14099: from __future__ import annotations
14100:
14101: import json
14102: import logging
14103: import sys
14104: from dataclasses import dataclass
14105: from enum import Enum
```

```
14106: from typing import Any, TypeVar
14107:
14108: try:
14109:     import psutil
14110:
14111:     PSUTIL_AVAILABLE = True
14112: except ImportError:
14113:     PSUTIL_AVAILABLE = False
14114:     psutil = None
14115:
14116: logger = logging.getLogger(__name__)
14117:
14118: T = TypeVar("T")
14119:
14120:
14121: class ExecutorType(Enum):
14122:     """Executor classification for memory limit configuration."""
14123:
14124:     ENTITY = "entity"
14125:     DAG = "dag"
14126:     CAUSAL_EFFECTS = "causal_effects"
14127:     SEMANTIC = "semantic"
14128:     FINANCIAL = "financial"
14129:     GENERIC = "generic"
14130:
14131:
14132: @dataclass
14133: class MemorySafetyConfig:
14134:     """Configuration for memory safety per executor type."""
14135:
14136:     entity_limit_mb: float = 1.0
14137:     dag_limit_mb: float = 5.0
14138:     causal_effects_limit_mb: float = 10.0
14139:     semantic_limit_mb: float = 2.0
14140:     financial_limit_mb: float = 2.0
14141:     generic_limit_mb: float = 5.0
14142:
14143:     memory_pressure_threshold_pct: float = 80.0
14144:     enable_pressure_detection: bool = True
14145:     enable_auto_sampling: bool = True
14146:     enable_auto_truncation: bool = True
14147:
14148:     max_list_elements: int = 1000
14149:     max_string_length: int = 100_000
14150:     max_dict_keys: int = 500
14151:
14152:     def get_limit_bytes(self, executor_type: ExecutorType) -> int:
14153:         """Get memory limit in bytes for executor type."""
14154:         limits = {
14155:             ExecutorType.ENTITY: self.entity_limit_mb,
14156:             ExecutorType.DAG: self.dag_limit_mb,
14157:             ExecutorType.CAUSAL_EFFECTS: self.causal_effects_limit_mb,
14158:             ExecutorType.SEMANTIC: self.semantic_limit_mb,
14159:             ExecutorType.FINANCIAL: self.financial_limit_mb,
14160:             ExecutorType.GENERIC: self.generic_limit_mb,
14161:         }
```

```
14162:         return int(limits[executor_type] * 1024 * 1024)
14163:
14164:
14165: @dataclass
14166: class MemoryMetrics:
14167:     """Memory usage metrics for monitoring."""
14168:
14169:     object_size_bytes: int
14170:     json_size_bytes: int
14171:     pressure_pct: float | None
14172:     was_truncated: bool
14173:     was_sampled: bool
14174:     fallback_strategy: str | None
14175:     elements_before: int | None
14176:     elements_after: int | None
14177:
14178:
14179: class MemoryPressureDetector:
14180:     """Detects system memory pressure to trigger fallback strategies."""
14181:
14182:     @staticmethod
14183:     def get_memory_pressure_pct() -> float | None:
14184:         """Get current system memory pressure percentage (0-100).
14185:
14186:         Returns None if psutil is not available.
14187:         """
14188:         if not PSUTIL_AVAILABLE:
14189:             return None
14190:
14191:         try:
14192:             memory = psutil.virtual_memory()
14193:             return memory.percent
14194:         except Exception as e:
14195:             logger.warning(f"Failed to read memory pressure: {e}")
14196:             return None
14197:
14198:     @staticmethod
14199:     def is_under_pressure(threshold_pct: float = 80.0) -> bool:
14200:         """Check if system is under memory pressure."""
14201:         pressure = MemoryPressureDetector.get_memory_pressure_pct()
14202:         if pressure is None:
14203:             return False
14204:         return pressure >= threshold_pct
14205:
14206:
14207: class ObjectSizeEstimator:
14208:     """Estimates size of Python objects including deep structures."""
14209:
14210:     @staticmethod
14211:     def estimate_object_size(obj: Any) -> int:
14212:         """Estimate size of Python object in bytes using sys.getsizeof.
14213:
14214:         For containers, recursively estimates contents up to reasonable depth.
14215:         """
14216:         size = sys.getsizeof(obj)
14217:
```

```
14218:         if isinstance(obj, dict):
14219:             size += sum(
14220:                 ObjectSizeEstimator.estimate_object_size(k)
14221:                 + ObjectSizeEstimator.estimate_object_size(v)
14222:                 for k, v in obj.items()
14223:             )
14224:         elif isinstance(obj, (list, tuple, set)):
14225:             size += sum(ObjectSizeEstimator.estimate_object_size(item) for item in obj)
14226:         elif hasattr(obj, "__dict__"):
14227:             size += ObjectSizeEstimator.estimate_object_size(obj.__dict__)
14228:
14229:     return size
14230:
14231: @staticmethod
14232: def estimate_json_size(obj: Any) -> int:
14233:     """Estimate serialized JSON size without full serialization.
14234:
14235:     Fast approximation:
14236:     - Strings: len(str) * 1.2 (accounting for escaping)
14237:     - Numbers: ~20 bytes
14238:     - Booleans/None: ~10 bytes
14239:     - Containers: sum of contents + overhead
14240:     """
14241:     if obj is None:
14242:         return 4
14243:
14244:     if isinstance(obj, bool):
14245:         return 5
14246:
14247:     if isinstance(obj, int):
14248:         return len(str(obj)) + 2
14249:
14250:     if isinstance(obj, float):
14251:         return 20
14252:
14253:     if isinstance(obj, str):
14254:         return int(len(obj) * 1.2) + 2
14255:
14256:     if isinstance(obj, (list, tuple)):
14257:         return 2 + sum(ObjectSizeEstimator.estimate_json_size(item) for item in obj)
14258:
14259:     if isinstance(obj, dict):
14260:         size = 2
14261:         for k, v in obj.items():
14262:             size += ObjectSizeEstimator.estimate_json_size(k) + 1
14263:             size += ObjectSizeEstimator.estimate_json_size(v) + 1
14264:         return size
14265:
14266:     try:
14267:         return len(json.dumps(obj))
14268:     except (TypeError, ValueError):
14269:         return sys.getsizeof(obj)
14270:
14271:
14272: class FallbackStrategy:
14273:     """Fallback strategies for handling objects exceeding size limits."""
```

```
14274:
14275:     @staticmethod
14276:     def sample_list(
14277:         items: list[T], max_elements: int, *, preserve_order: bool = True
14278:     ) -> list[T]:
14279:         """Sample list to max_elements using systematic sampling.
14280:
14281:             Args:
14282:                 items: List to sample
14283:                 max_elements: Maximum elements to keep
14284:                 preserve_order: Whether to maintain original order
14285:
14286:             Returns:
14287:                 Sampled list
14288: """
14289:     if len(items) <= max_elements:
14290:         return items
14291:
14292:     if preserve_order:
14293:         step = len(items) / max_elements
14294:         indices = [int(i * step) for i in range(max_elements)]
14295:         return [items[i] for i in indices]
14296:
14297:     import random
14298:
14299:     return random.sample(items, max_elements)
14300:
14301:     @staticmethod
14302:     def truncate_string(s: str, max_length: int) -> str:
14303:         """Truncate string to max_length with ellipsis."""
14304:         if len(s) <= max_length:
14305:             return s
14306:         return s[:max_length - 3] + "..."
14307:
14308:     @staticmethod
14309:     def truncate_dict(d: dict[str, Any], max_keys: int) -> dict[str, Any]:
14310:         """Truncate dictionary to max_keys, preserving most important keys."""
14311:         if len(d) <= max_keys:
14312:             return d
14313:
14314:         priority_keys = ["id", "name", "type", "label", "value", "score", "confidence"]
14315:
14316:         result = {}
14317:         for key in priority_keys:
14318:             if key in d and len(result) < max_keys:
14319:                 result[key] = d[key]
14320:
14321:         remaining = max_keys - len(result)
14322:         for key in d:
14323:             if key not in result and remaining > 0:
14324:                 result[key] = d[key]
14325:                 remaining -= 1
14326:
14327:         return result
14328:
14329:     @staticmethod
```

```
14330:     def apply_recursive_truncation(
14331:         obj: Any, config: MemorySafetyConfig, depth: int = 0, max_depth: int = 10
14332:     ) -> tuple[Any, bool]:
14333:         """Recursively apply truncation strategies to object tree.
14334:
14335:         Returns:
14336:             (truncated_object, was_modified)
14337:         """
14338:         if depth > max_depth:
14339:             return obj, False
14340:
14341:         modified = False
14342:
14343:         if isinstance(obj, str):
14344:             if len(obj) > config.max_string_length:
14345:                 obj = FallbackStrategy.truncate_string(obj, config.max_string_length)
14346:                 modified = True
14347:
14348:         elif isinstance(obj, list):
14349:             if len(obj) > config.max_list_elements:
14350:                 obj = FallbackStrategy.sample_list(obj, config.max_list_elements)
14351:                 modified = True
14352:
14353:             new_items = []
14354:             for item in obj:
14355:                 new_item, item_modified = FallbackStrategy.apply_recursive_truncation(
14356:                     item, config, depth + 1, max_depth
14357:                 )
14358:                 new_items.append(new_item)
14359:                 modified = modified or item_modified
14360:             obj = new_items
14361:
14362:         elif isinstance(obj, dict):
14363:             if len(obj) > config.max_dict_keys:
14364:                 obj = FallbackStrategy.truncate_dict(obj, config.max_dict_keys)
14365:                 modified = True
14366:
14367:             new_dict = {}
14368:             for k, v in obj.items():
14369:                 new_v, v_modified = FallbackStrategy.apply_recursive_truncation(
14370:                     v, config, depth + 1, max_depth
14371:                 )
14372:                 new_dict[k] = new_v
14373:                 modified = modified or v_modified
14374:             obj = new_dict
14375:
14376:         return obj, modified
14377:
14378:
14379: class MemorySafetyGuard:
14380:     """Main guard for memory-safe object processing."""
14381:
14382:     def __init__(self, config: MemorySafetyConfig | None = None):
14383:         self.config = config or MemorySafetyConfig()
14384:         self.metrics: list[MemoryMetrics] = []
14385:
```

```
14386:     def check_and_process(
14387:         self, obj: Any, executor_type: ExecutorType, label: str = "object"
14388:     ) -> tuple[Any, MemoryMetrics]:
14389:         """Check object size and apply fallback strategies if needed.
14390:
14391:         Args:
14392:             obj: Object to check
14393:             executor_type: Type of executor processing this object
14394:             label: Human-readable label for logging
14395:
14396:         Returns:
14397:             (processed_object, metrics)
14398:         """
14399:         obj_size = ObjectSizeEstimator.estimate_object_size(obj)
14400:         json_size = ObjectSizeEstimator.estimate_json_size(obj)
14401:         limit_bytes = self.config.get_limit_bytes(executor_type)
14402:
14403:         pressure_pct = None
14404:         if self.config.enable_pressure_detection:
14405:             pressure_pct = MemoryPressureDetector.get_memory_pressure_pct()
14406:
14407:         was_truncated = False
14408:         was_sampled = False
14409:         fallback_strategy = None
14410:         elements_before = self._count_elements(obj)
14411:
14412:         under_pressure = (
14413:             pressure_pct is not None
14414:             and pressure_pct >= self.config.memory_pressure_threshold_pct
14415:         )
14416:
14417:         if obj_size > limit_bytes or json_size > limit_bytes or under_pressure:
14418:             reason = []
14419:             if obj_size > limit_bytes:
14420:                 reason.append(
14421:                     f"object size {obj_size / (1024*1024):.2f}MB exceeds {limit_bytes / (1024*1024):.2f}MB"
14422:                 )
14423:             if json_size > limit_bytes:
14424:                 reason.append(
14425:                     f"JSON size {json_size / (1024*1024):.2f}MB exceeds {limit_bytes / (1024*1024):.2f}MB"
14426:                 )
14427:             if under_pressure:
14428:                 reason.append(
14429:                     f"memory pressure {pressure_pct:.1f}% >= {self.config.memory_pressure_threshold_pct}%""
14430:                 )
14431:
14432:             logger.warning(
14433:                 f"Memory safety triggered for {label} ({executor_type.value}): {'; '.join(reason)}"
14434:             )
14435:
14436:             if self.config.enable_auto_truncation:
14437:                 obj, was_truncated = FallbackStrategy.apply_recursive_truncation(
14438:                     obj, self.config
14439:                 )
14440:                 fallback_strategy = "truncation"
14441:
```

```
14442:         obj_size = ObjectSizeEstimator.estimate_object_size(obj)
14443:         json_size = ObjectSizeEstimator.estimate_json_size(obj)
14444:         logger.info(
14445:             f"Applied truncation to {label}: "
14446:             f"new size {obj_size / (1024*1024):.2f}MB object, "
14447:             f"{json_size / (1024*1024):.2f}MB JSON"
14448:         )
14449:
14450:     elements_after = self._count_elements(obj)
14451:
14452:     metrics = MemoryMetrics(
14453:         object_size_bytes=obj_size,
14454:         json_size_bytes=json_size,
14455:         pressure_pct=pressure_pct,
14456:         was_truncated=was_truncated,
14457:         was_sampled=was_sampled,
14458:         fallback_strategy=fallback_strategy,
14459:         elements_before=elements_before,
14460:         elements_after=elements_after,
14461:     )
14462:
14463:     self.metrics.append(metrics)
14464:     return obj, metrics
14465:
14466: def _count_elements(self, obj: Any) -> int | None:
14467:     """Count elements in container objects."""
14468:     if isinstance(obj, (list, tuple)):
14469:         return len(obj)
14470:     if isinstance(obj, dict):
14471:         return len(obj)
14472:     return None
14473:
14474: def get_metrics_summary(self) -> dict[str, Any]:
14475:     """Get summary of all memory operations."""
14476:     if not self.metrics:
14477:         return {
14478:             "total_operations": 0,
14479:             "truncations": 0,
14480:             "samplings": 0,
14481:             "avg_object_size_mb": 0.0,
14482:             "avg_json_size_mb": 0.0,
14483:             "max_object_size_mb": 0.0,
14484:             "max_json_size_mb": 0.0,
14485:         }
14486:
14487:     return {
14488:         "total_operations": len(self.metrics),
14489:         "truncations": sum(1 for m in self.metrics if m.was_truncated),
14490:         "samplings": sum(1 for m in self.metrics if m.was_sampled),
14491:         "avg_object_size_mb": sum(m.object_size_bytes for m in self.metrics)
14492:             / len(self.metrics)
14493:             / (1024 * 1024),
14494:         "avg_json_size_mb": sum(m.json_size_bytes for m in self.metrics)
14495:             / len(self.metrics)
14496:             / (1024 * 1024),
14497:         "max_object_size_mb": max(m.object_size_bytes for m in self.metrics)
```

```
14498:         / (1024 * 1024),
14499:         "max_json_size_mb": max(m.json_size_bytes for m in self.metrics)
14500:         / (1024 * 1024),
14501:         "pressure_samples": [
14502:             m.pressure_pct for m in self.metrics if m.pressure_pct is not None
14503:         ],
14504:     }
14505:
14506:
14507: def create_default_guard() -> MemorySafetyGuard:
14508:     """Create memory safety guard with default configuration."""
14509:     return MemorySafetyGuard(MemorySafetyConfig())
14510:
14511:
14512: __all__ = [
14513:     "ExecutorType",
14514:     "MemorySafetyConfig",
14515:     "MemoryMetrics",
14516:     "MemoryPressureDetector",
14517:     "ObjectSizeEstimator",
14518:     "FallbackStrategy",
14519:     "MemorySafetyGuard",
14520:     "create_default_guard",
14521: ]
14522:
14523:
14524:
14525: =====
14526: FILE: src/farfan_pipeline/core/orchestrator/method_registry.py
14527: =====
14528:
14529: """Method Registry with lazy instantiation and injection pattern.
14530:
14531: This module implements a method injection factory that:
14532: 1. Loads only the methods needed (not full classes)
14533: 2. Instantiates classes lazily (only when first method is called)
14534: 3. Caches instances for reuse
14535: 4. Isolates errors per method (failures don't cascade)
14536: 5. Allows direct function injection (bypassing classes)
14537:
14538: Architecture:
14539:     MethodRegistry
14540:         \224\234\224\200 _class_paths: mapping of class names to import paths
14541:         \224\234\224\200 _instance_cache: lazily instantiated class instances
14542:         \224\234\224\200 _direct_methods: directly injected functions
14543:         \224\224\224\200 get_method(): returns callable for (class_name, method_name)
14544:
14545: Benefits:
14546: - No upfront class loading (lightweight imports)
14547: - Failed classes don't block working methods
14548: - Direct function injection for custom implementations
14549: - Instance reuse through caching
14550: """
14551: from __future__ import annotations
14552:
14553: import logging
```

```
14554: import threading
14555: from importlib import import_module
14556: from typing import Any, Callable
14557:
14558: logger = logging.getLogger(__name__)
14559:
14560:
14561: class MethodRegistryError(RuntimeError):
14562:     """Raised when a method cannot be retrieved."""
14563:
14564:
14565: class MethodRegistry:
14566:     """Registry for lazy method injection without full class instantiation."""
14567:
14568:     def __init__(self, class_paths: dict[str, str] | None = None) -> None:
14569:         """Initialize the method registry.
14570:
14571:             Args:
14572:                 class_paths: Optional mapping of class names to import paths.
14573:                             If None, uses default paths from class_registry.
14574:
14575:             # Import class paths from existing registry
14576:             if class_paths is None:
14577:                 from farfan_pipeline.core.orchestrator.class_registry import get_class_paths
14578:                 class_paths = dict(get_class_paths())
14579:
14580:             self._class_paths = class_paths
14581:             self._instance_cache: dict[str, Any] = {}
14582:             self._direct_methods: dict[tuple[str, str], Callable[..., Any]] = {}
14583:             self._failed_classes: set[str] = set()
14584:             self._lock = threading.Lock()
14585:
14586:             # Special instantiation rules (from original MethodExecutor)
14587:             self._special_instantiation: dict[str, Callable[[type], Any]] = {}
14588:
14589:     def inject_method(
14590:         self,
14591:         class_name: str,
14592:         method_name: str,
14593:         method: Callable[..., Any],
14594:     ) -> None:
14595:         """Directly inject a method without needing a class.
14596:
14597:             This allows bypassing class instantiation entirely.
14598:
14599:             Args:
14600:                 class_name: Virtual class name for routing
14601:                 method_name: Method name
14602:                 method: Callable to inject
14603:
14604:             key = (class_name, method_name)
14605:             self._direct_methods[key] = method
14606:             logger.info(
14607:                 "method_injected_directly",
14608:                 class_name=class_name,
14609:                 method_name=method_name,
```

```
14610:         )
14611:
14612:     def register_instantiation_rule(
14613:         self,
14614:         class_name: str,
14615:         instantiator: Callable[[type], Any],
14616:     ) -> None:
14617:         """Register special instantiation logic for a class.
14618:
14619:         Args:
14620:             class_name: Class name requiring special instantiation
14621:             instantiator: Function that takes class type and returns instance
14622:
14623:         self._special_instantiation[class_name] = instantiator
14624:         logger.debug(
14625:             "instantiation_rule_registered",
14626:             class_name=class_name,
14627:         )
14628:
14629:     def _load_class(self, class_name: str) -> type:
14630:         """Load a class type from import path.
14631:
14632:         Args:
14633:             class_name: Name of class to load
14634:
14635:         Returns:
14636:             Class type
14637:
14638:         Raises:
14639:             MethodRegistryError: If class cannot be loaded
14640:
14641:         if class_name not in self._class_paths:
14642:             raise MethodRegistryError(
14643:                 f"Class '{class_name}' not found in registry paths"
14644:             )
14645:
14646:         path = self._class_paths[class_name]
14647:         module_name, _, attr_name = path.rpartition(".")
14648:
14649:         if not module_name:
14650:             raise MethodRegistryError(
14651:                 f"Invalid path for '{class_name}': {path}"
14652:             )
14653:
14654:         try:
14655:             module = import_module(module_name)
14656:             cls = getattr(module, attr_name)
14657:
14658:             if not isinstance(cls, type):
14659:                 raise MethodRegistryError(
14660:                     f"'{class_name}' is not a class: {type(cls).__name__}"
14661:                 )
14662:
14663:             return cls
14664:
14665:         except ImportError as exc:
```

```
14666:         raise MethodRegistryError(
14667:             f"Cannot import class '{class_name}' from {path}: {exc}"
14668:         ) from exc
14669:     except AttributeError as exc:
14670:         raise MethodRegistryError(
14671:             f"Class '{attr_name}' not found in module {module_name}: {exc}"
14672:         ) from exc
14673:
14674:     def _instantiate_class(self, class_name: str, cls: type) -> Any:
14675:         """Instantiate a class using special rules or default constructor.
14676:
14677:         Args:
14678:             class_name: Name of class (for special rule lookup)
14679:             cls: Class type to instantiate
14680:
14681:         Returns:
14682:             Instance of the class
14683:
14684:         Raises:
14685:             MethodRegistryError: If instantiation fails
14686:         """
14687:         # Use special instantiation rule if registered
14688:         if class_name in self._special_instantiation:
14689:             try:
14690:                 instantiator = self._special_instantiation[class_name]
14691:                 instance = instantiator(cls)
14692:                 logger.debug(
14693:                     "class_instantiated_with_special_rule",
14694:                     class_name=class_name,
14695:                 )
14696:                 return instance
14697:             except Exception as exc:
14698:                 raise MethodRegistryError(
14699:                     f"Special instantiation failed for '{class_name}': {exc}"
14700:                 ) from exc
14701:
14702:         # Default instantiation (no-args constructor)
14703:         try:
14704:             instance = cls()
14705:             logger.debug(
14706:                 "class_instantiated_default",
14707:                 class_name=class_name,
14708:             )
14709:             return instance
14710:         except Exception as exc:
14711:             raise MethodRegistryError(
14712:                 f"Default instantiation failed for '{class_name}': {exc}"
14713:             ) from exc
14714:
14715:     def _get_instance(self, class_name: str) -> Any:
14716:         """Get or create instance of a class (lazy + cached).
14717:
14718:         Args:
14719:             class_name: Name of class to instantiate
14720:
14721:         Returns:
```

```
14722:             Instance of the class
14723:
14724:             Raises:
14725:                 MethodRegistryError: If class cannot be instantiated
14726:
14727:             """"
14728:             # Check if already failed
14729:             if class_name in self._failed_classes:
14730:                 raise MethodRegistryError(
14731:                     f"Class '{class_name}' previously failed to instantiate"
14732:                 )
14733:
14734:             # Use a lock to ensure thread-safe instantiation
14735:             with self._lock:
14736:                 # Double-check if another thread instantiated it while waiting for the lock
14737:                 if class_name in self._instance_cache:
14738:                     return self._instance_cache[class_name]
14739:
14740:             # Load and instantiate class
14741:             try:
14742:                 cls = self._load_class(class_name)
14743:                 instance = self._instantiate_class(class_name, cls)
14744:                 self._instance_cache[class_name] = instance
14745:                 logger.info(
14746:                     "class_instantiated_lazy",
14747:                     class_name=class_name,
14748:                 )
14749:                 return instance
14750:
14751:             except MethodRegistryError:
14752:                 # Mark as failed to avoid repeated attempts
14753:                 self._failed_classes.add(class_name)
14754:
14755:             def get_method(
14756:                 self,
14757:                 class_name: str,
14758:                 method_name: str,
14759:             ) -> Callable[..., Any]:
14760:                 """Get method callable with lazy instantiation.
14761:
14762:                 This is the main entry point for retrieving methods.
14763:
14764:                 Args:
14765:                     class_name: Name of class containing the method
14766:                     method_name: Name of method to retrieve
14767:
14768:                 Returns:
14769:                     Callable method (bound or injected)
14770:
14771:                 Raises:
14772:                     MethodRegistryError: If method cannot be retrieved
14773:
14774:                     """"
14775:                     # Check for directly injected method first
14776:                     key = (class_name, method_name)
14777:                     if key in self._direct_methods:
14778:                         logger.debug(
```

```
14778:             "method_retrieved_direct",
14779:             class_name=class_name,
14780:             method_name=method_name,
14781:         )
14782:         return self._direct_methods[key]
14783:
14784:     # Get instance (lazy) and retrieve method
14785:     try:
14786:         instance = self._get_instance(class_name)
14787:         method = getattr(instance, method_name)
14788:
14789:         if not callable(method):
14790:             raise MethodRegistryError(
14791:                 f'{class_name}.{method_name}' is not callable"
14792:             )
14793:
14794:         logger.debug(
14795:             "method_retrieved_from_instance",
14796:             class_name=class_name,
14797:             method_name=method_name,
14798:         )
14799:         return method
14800:
14801:     except AttributeError as exc:
14802:         raise MethodRegistryError(
14803:             f"Method '{method_name}' not found on class '{class_name}'"
14804:         ) from exc
14805:
14806:     def has_method(self, class_name: str, method_name: str) -> bool:
14807:         """Check if a method is available (without instantiating).
14808:
14809:         Args:
14810:             class_name: Name of class
14811:             method_name: Name of method
14812:
14813:         Returns:
14814:             True if method exists (or is directly injected)
14815:             """
14816:             # Check direct injection
14817:             key = (class_name, method_name)
14818:             if key in self._direct_methods:
14819:                 return True
14820:
14821:             # Check if class is known and not failed
14822:             if class_name in self._failed_classes:
14823:                 return False
14824:
14825:             if class_name not in self._class_paths:
14826:                 return False
14827:
14828:             # If instance exists, check method
14829:             if class_name in self._instance_cache:
14830:                 instance = self._instance_cache[class_name]
14831:                 return hasattr(instance, method_name)
14832:
14833:             # Otherwise, assume it exists (lazy check)
```

```
14834:         # Full validation happens on first get_method() call
14835:         return True
14836:
14837:     def get_stats(self) -> dict[str, Any]:
14838:         """Get registry statistics.
14839:
14840:         Returns:
14841:             Dictionary with registry stats
14842:         """
14843:         return {
14844:             "total_classes_registered": len(self._class_paths),
14845:             "instantiated_classes": len(self._instance_cache),
14846:             "failed_classes": len(self._failed_classes),
14847:             "direct_methods_injected": len(self._direct_methods),
14848:             "instantiated_class_names": list(self._instance_cache.keys()),
14849:             "failed_class_names": list(self._failed_classes),
14850:         }
14851:
14852:
14853: def setup_default_instantiation_rules(registry: MethodRegistry) -> None:
14854:     """Setup default special instantiation rules.
14855:
14856:     These rules replicate the logic from the original MethodExecutor
14857:     for classes that need non-default instantiation.
14858:
14859:     Args:
14860:         registry: MethodRegistry to configure
14861:     """
14862:     # MunicipalOntology - shared instance pattern
14863:     ontology_instance = None
14864:
14865:     def instantiate_ontology(cls: type) -> Any:
14866:         nonlocal ontology_instance
14867:         if ontology_instance is None:
14868:             ontology_instance = cls()
14869:         return ontology_instance
14870:
14871:     registry.register_instantiation_rule("MunicipalOntology", instantiate_ontology)
14872:
14873:     # SemanticAnalyzer, PerformanceAnalyzer, TextMiningEngine - need ontology
14874:     def instantiate_with_ontology(cls: type) -> Any:
14875:         if ontology_instance is None:
14876:             raise MethodRegistryError(
14877:                 f"Cannot instantiate {cls.__name__}: MunicipalOntology not available"
14878:             )
14879:         return cls(ontology_instance)
14880:
14881:     for class_name in ["SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine"]:
14882:         registry.register_instantiation_rule(class_name, instantiate_with_ontology)
14883:
14884:     # PolicyTextProcessor - needs ProcessorConfig
14885:     def instantiate_policy_processor(cls: type) -> Any:
14886:         try:
14887:             from farfan_pipeline.processing.policy_processor import ProcessorConfig
14888:             return cls(ProcessorConfig())
14889:         except ImportError as exc:
```



```
15001:             fqn = f"{class_name}.{method_name}"
15002:
15003:             # Basic signature extraction
15004:             args = [arg.arg for arg in item.args.args]
15005:             signature = f"({', '.join(args)})"
15006:             # A more advanced version would parse type hints if they exist
15007:
15008:             source_truth[fqn] = {
15009:                 "exists": True,
15010:                 "file": file_path,
15011:                 "line": item.lineno,
15012:                 "signature": signature,
15013:             }
15014:         return source_truth
15015:
15016: if __name__ == "__main__":
15017:     validator = MethodSourceValidator()
15018:
15019:     # 1. Generate the ground-truth map
15020:     source_truth_map = validator.generate_source_truth_map()
15021:     output_path = "method_source_truth.json"
15022:     with open(output_path, "w") as f:
15023:         json.dump(source_truth_map, f, indent=4)
15024:     print(f"Generated source truth map at {output_path}")
15025:
15026:     # 2. Validate executor methods
15027:     validation_report = validator.validate_executor_methods()
15028:     report_path = "executor_validation_report.json"
15029:     with open(report_path, "w") as f:
15030:         json.dump(validation_report, f, indent=4)
15031:     print(f"Validation report generated at {report_path}")
15032:
15033:     print("\nValidation Summary:")
15034:     print(f" - Valid methods: {len(validation_report['valid'])}")
15035:     print(f" - Missing methods: {len(validation_report['missing'])}")
15036:     if validation_report['missing']:
15037:         print("\nMissing methods:")
15038:         for method in validation_report['missing']:
15039:             print(f" - {method}")
15040:
15041:
15042:
15043: =====
15044: FILE: src/farfan_pipeline/core/orchestrator/phase6_validation.py
15045: =====
15046:
15047: """Phase 6: Schema Validation Pipeline - Four Subphase Architecture.
15048:
15049: This module implements Phase 6 as a complete validation pipeline with four subphases:
15050:
15051: Phase 6.1: Classification & Extraction
15052:     - Extracts question_global via bracket notation (question["question_global"])
15053:     - Extracts expected_elements via get method with None handling
15054:     - Classifies types using isinstance checks in None-list-dict-invalid order
15055:     - Stores classification tuple before any iteration occurs
15056:
```

```
15057: Phase 6.2: Structural Validation
15058:     - Checks invalid types first with human-readable type names
15059:     - Enforces homogeneity allowing None compatibility
15060:     - Validates list length equality and dict key set equality
15061:     - Uses symmetric difference computation for dict key validation
15062:     - Returns silently on dual-None without logging
15063:
15064: Phase 6.3: Semantic Validation
15065:     - Iterates deterministically via enumerate-zip for lists and sorted keys for dicts
15066:     - Extracts type-required-minimum fields with get defaults
15067:     - Implements asymmetric required implication as not-q-or-c boolean expression
15068:     - Enforces c-min-greater-equal-q-min threshold ordering
15069:     - Returns validated element count
15070:
15071: Phase 6.4: Orchestrator
15072:     - Invokes structural then semantic layers in sequence
15073:     - Captures element count return value
15074:     - Emits debug log with has_required_fields and has_minimum_thresholds computed
15075:         via any-element-iteration
15076:     - Logs info warning for None chunk schema with non-None question schema
15077:     - Integrates into build_with_chunk_matrix loop after Phase 5 before construct_task
15078:     - Allows TypeError-ValueError propagation to outer handler without try-except wrapping
15079:
15080: Architecture:
15081:     Phase 6.1 à\206\222 Phase 6.2 à\206\222 Phase 6.3 à\206\222 Phase 6.4
15082:     (Sequential root) (Structural) (Semantic) (Synchronization barrier)
15083:
15084: Parallelization:
15085:     - Phase 6.1: Sequential root (extracts and classifies)
15086:     - Phase 6.2-6.3: Concurrency potential (independent validation layers)
15087:     - Phase 6.4: Synchronization barrier (aggregates results)
15088: """
15089:
15090: from __future__ import annotations
15091:
15092: import logging
15093: from typing import Any
15094:
15095: logger = logging.getLogger(__name__)
15096:
15097: MAX_QUESTION_GLOBAL = 999
15098:
15099:
15100: def _classify_expected_elements_type(value: Any) -> str: # noqa: ANN401
15101:     """Phase 6.1: Classify expected_elements type using isinstance checks.
15102:
15103:     Performs type classification in None-list-dict-invalid order via identity
15104:     test for None, then isinstance checks for list and dict, with any other
15105:     type classified as invalid.
15106:
15107:     Args:
15108:         value: Value to classify (expected_elements from question or chunk)
15109:
15110:     Returns:
15111:         Type classification string: "none", "list", "dict", or "invalid"
15112:
```

```

15113:     Classification Order:
15114:         1. None via identity test (value is None)
15115:         2. list via isinstance(value, list)
15116:         3. dict via isinstance(value, dict)
15117:         4. invalid for all other types
15118:     """
15119:     if value is None:
15120:         return "none"
15121:     elif isinstance(value, list):
15122:         return "list"
15123:     elif isinstance(value, dict):
15124:         return "dict"
15125:     else:
15126:         return "invalid"
15127:
15128:
15129: def _extract_and_classify_schemas(
15130:     question: dict[str, Any],
15131:     chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
15132:     question_id: str,
15133: ) -> tuple[int, Any, Any, str, str]: # noqa: ANN401
15134:     """Phase 6.1: Extract question_global and expected_elements, classify types.
15135:
15136:     Extracts question_global via bracket notation (question["question_global"])
15137:     and expected_elements via get method with None default. Classifies both
15138:     schema types and stores classification tuple before any iteration occurs.
15139:
15140:     Args:
15141:         question: Question dictionary from questionnaire
15142:         chunk_expected_elements: expected_elements from chunk routing result
15143:         question_id: Question identifier for error reporting
15144:
15145:     Returns:
15146:         Tuple of (question_global, question_schema, chunk_schema,
15147:                   question_type, chunk_type)
15148:
15149:     Raises:
15150:         ValueError: If question_global is missing, invalid type, or out of range
15151:     """
15152:     # Extract question_global via bracket notation
15153:     try:
15154:         question_global = question["question_global"]
15155:     except KeyError as e:
15156:         raise ValueError(
15157:             f"Schema validation failure for question {question_id}: "
15158:             "question_global field is required but missing"
15159:         ) from e
15160:
15161:     # Validate question_global
15162:     if not isinstance(question_global, int):
15163:         raise ValueError(
15164:             f"Schema validation failure for question {question_id}: "
15165:             f"question_global must be an integer, got {type(question_global).__name__}"
15166:         )
15167:
15168:     if not (0 <= question_global <= MAX_QUESTION_GLOBAL):

```

```

15169:         raise ValueError(
15170:             f"Schema validation failure for question {question_id}: "
15171:             f"question_global must be between 0 and {MAX_QUESTION_GLOBAL} inclusive, got {question_global}"
15172:         )
15173:
15174:     # Extract expected_elements via get method with None handling
15175:     question_schema = question.get("expected_elements")
15176:     chunk_schema = chunk_expected_elements
15177:
15178:     # Classify types using isinstance checks in None-list-dict-invalid order
15179:     question_type = _classify_expected_elements_type(question_schema)
15180:     chunk_type = _classify_expected_elements_type(chunk_schema)
15181:
15182:     # Store classification tuple before any iteration occurs
15183:     return question_global, question_schema, chunk_schema, question_type, chunk_type
15184:
15185:
15186: def _validate_structural_compatibility(
15187:     question_schema: Any, # noqa: ANN401
15188:     chunk_schema: Any, # noqa: ANN401
15189:     question_type: str,
15190:     chunk_type: str,
15191:     question_id: str,
15192:     correlation_id: str,
15193: ) -> None:
15194:     """Phase 6.2: Validate structural compatibility with type homogeneity checks.
15195:
15196:     Checks invalid types first with human-readable type names, enforces
15197:     homogeneity allowing None compatibility, validates list length equality
15198:     and dict key set equality with symmetric difference computation, and
15199:     returns silently on dual-None without logging.
15200:
15201:     Args:
15202:         question_schema: expected_elements from question
15203:         chunk_schema: expected_elements from chunk
15204:         question_type: Classified type of question schema
15205:         chunk_type: Classified type of chunk schema
15206:         question_id: Question identifier for error messages
15207:         correlation_id: Correlation ID for distributed tracing
15208:
15209:     Raises:
15210:         TypeError: If either schema has invalid type (not list, dict, or None)
15211:         ValueError: If schemas have heterogeneous types (not allowing None),
15212:                     list length mismatch, or dict key set mismatch
15213:
15214:     Returns:
15215:         None (returns silently on dual-None or successful validation)
15216:     """
15217:     # Check invalid types first with human-readable type names
15218:     if question_type == "invalid":
15219:         raise TypeError(
15220:             f"Schema validation failure for question {question_id}: "
15221:             f"expected_elements from question has invalid type "
15222:             f"{type(question_schema).__name__}, expected list, dict, or None "
15223:             f"[correlation_id={correlation_id}]"
15224:         )

```

```
15225:  
15226:     if chunk_type == "invalid":  
15227:         raise TypeError(  
15228:             f"Schema validation failure for question {question_id}: "  
15229:             f"expected_elements from chunk has invalid type "  
15230:             f"{{type(chunk_schema).__name__}}, expected list, dict, or None "  
15231:             f"[correlation_id={correlation_id}]"  
15232:     )  
15233:  
15234: # Return silently on dual-None without logging  
15235: if question_type == "none" and chunk_type == "none":  
15236:     return  
15237:  
15238: # Enforce homogeneity allowing None compatibility  
15239: # None is compatible with any type, but non-None types must match  
15240: if question_type not in ("none", chunk_type) and chunk_type != "none":  
15241:     raise ValueError(  
15242:         f"Schema validation failure for question {question_id}: "  
15243:         f"heterogeneous types detected (question has {question_type}, "  
15244:         f"chunk has {chunk_type}) [correlation_id={correlation_id}]"  
15245:     )  
15246:  
15247: # Validate list length equality  
15248: if question_type == "list" and chunk_type == "list":  
15249:     question_len = len(question_schema)  
15250:     chunk_len = len(chunk_schema)  
15251:     if question_len != chunk_len:  
15252:         raise ValueError(  
15253:             f"Schema validation failure for question {question_id}: "  
15254:             f"list length mismatch (question has {question_len} elements, "  
15255:             f"chunk has {chunk_len} elements) [correlation_id={correlation_id}]"  
15256:     )  
15257:  
15258: # Validate dict key set equality with symmetric difference computation  
15259: if question_type == "dict" and chunk_type == "dict":  
15260:     question_keys = set(question_schema.keys())  
15261:     chunk_keys = set(chunk_schema.keys())  
15262:  
15263:     # Compute symmetric difference  
15264:     symmetric_diff = question_keys ^ chunk_keys  
15265:  
15266:     if symmetric_diff:  
15267:         missing_in_chunk = question_keys - chunk_keys  
15268:         extra_in_chunk = chunk_keys - question_keys  
15269:  
15270:         details = []  
15271:         if missing_in_chunk:  
15272:             details.append(f"missing in chunk: {sorted(missing_in_chunk)}")  
15273:         if extra_in_chunk:  
15274:             details.append(f"extra in chunk: {sorted(extra_in_chunk)}")  
15275:  
15276:     raise ValueError(  
15277:         f"Schema validation failure for question {question_id}: "  
15278:         f"dict key set mismatch ({', '.join(details)}) "  
15279:         f"[correlation_id={correlation_id}]"  
15280:     )
```

```
15281:  
15282:  
15283: def _validate_semantic_constraints(  
15284:     question_schema: Any, # noqa: ANN401  
15285:     chunk_schema: Any, # noqa: ANN401  
15286:     question_type: str,  
15287:     chunk_type: str,  
15288:     provisional_task_id: str,  
15289:     question_id: str,  
15290:     chunk_id: str,  
15291:     correlation_id: str,  
15292: ) -> int:  
15293:     """Phase 6.3: Validate semantic constraints and return validated element count.  
15294:  
15295:     Iterates deterministically via enumerate-zip for lists and sorted keys for  
15296:     dicts, extracts type-required-minimum fields with get defaults, implements  
15297:     asymmetric required implication as not-q-or-c boolean expression, enforces  
15298:     c-min-greater-equal-q-min threshold ordering, and returns validated element  
15299:     count.  
15300:  
15301:     Args:  
15302:         question_schema: expected_elements from question  
15303:         chunk_schema: expected_elements from chunk  
15304:         question_type: Classified type of question schema  
15305:         chunk_type: Classified type of chunk schema  
15306:         provisional_task_id: Task ID for error reporting  
15307:         question_id: Question identifier for error messages  
15308:         chunk_id: Chunk identifier for error messages  
15309:         correlation_id: Correlation ID for distributed tracing  
15310:  
15311:     Returns:  
15312:         Validated element count (number of elements validated)  
15313:  
15314:     Raises:  
15315:         ValueError: If required field implication violated or threshold ordering violated  
15316:  
15317:     Semantic Constraints:  
15318:         - Asymmetric required implication: not q_required or c_required  
15319:         - Threshold ordering: c_minimum >= q_minimum  
15320:     """  
15321:     validated_count = 0  
15322:  
15323:     # Iterate deterministically via enumerate-zip for lists  
15324:     if question_type == "list" and chunk_type == "list":  
15325:         for idx, (q_elem, c_elem) in enumerate(  
15326:             zip(question_schema, chunk_schema, strict=True)  
15327:         ):  
15328:             if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):  
15329:                 continue  
15330:  
15331:             # Extract type-required-minimum fields with get defaults  
15332:             element_type = q_elem.get("type", f"element_at_index_{idx}")  
15333:             q_required = q_elem.get("required", False)  
15334:             c_required = c_elem.get("required", False)  
15335:             q_minimum = q_elem.get("minimum", 0)  
15336:             c_minimum = c_elem.get("minimum", 0)
```

```
15337:
15338:    # Implement asymmetric required implication as not-q-or-c boolean expression
15339:    if q_required and not c_required:
15340:        raise ValueError(
15341:            f"Task {provisional_task_id}: Required field implication violation "
15342:            f"at index {idx}: element type '{element_type}' is required in "
15343:            f"question but marked as optional in chunk "
15344:            f"[question_id={question_id}, chunk_id={chunk_id}, "
15345:            f"correlation_id={correlation_id}]"
15346:        )
15347:
15348:    # Enforce c-min-greater-equal-q-min threshold ordering
15349:    if (
15350:        isinstance(q_minimum, int | float)
15351:        and isinstance(c_minimum, int | float)
15352:        and c_minimum < q_minimum
15353:    ):
15354:        raise ValueError(
15355:            f"Task {provisional_task_id}: Threshold ordering violation "
15356:            f"at index {idx}: element type '{element_type}' has chunk "
15357:            f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
15358:            f"[question_id={question_id}, chunk_id={chunk_id}, "
15359:            f"correlation_id={correlation_id}]"
15360:        )
15361:
15362:    validated_count += 1
15363:
15364:    # Iterate deterministically via sorted keys for dicts
15365:    elif question_type == "dict" and chunk_type == "dict":
15366:        common_keys = set(question_schema.keys()) & set(chunk_schema.keys())
15367:
15368:        for key in sorted(common_keys):
15369:            q_elem = question_schema[key]
15370:            c_elem = chunk_schema[key]
15371:
15372:            if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
15373:                continue
15374:
15375:            # Extract type-required-minimum fields with get defaults
15376:            element_type = q_elem.get("type", key)
15377:            q_required = q_elem.get("required", False)
15378:            c_required = c_elem.get("required", False)
15379:            q_minimum = q_elem.get("minimum", 0)
15380:            c_minimum = c_elem.get("minimum", 0)
15381:
15382:            # Implement asymmetric required implication as not-q-or-c boolean expression
15383:            if q_required and not c_required:
15384:                raise ValueError(
15385:                    f"Task {provisional_task_id}: Required field implication violation "
15386:                    f"for key '{key}': element type '{element_type}' is required in "
15387:                    f"question but marked as optional in chunk "
15388:                    f"[question_id={question_id}, chunk_id={chunk_id}, "
15389:                    f"correlation_id={correlation_id}]"
15390:                )
15391:
15392:    # Enforce c-min-greater-equal-q-min threshold ordering
```

```
15393:         if (
15394:             isinstance(q_minimum, int | float)
15395:             and isinstance(c_minimum, int | float)
15396:             and c_minimum < q_minimum
15397:         ):
15398:             raise ValueError(
15399:                 f"Task {provisional_task_id}: Threshold ordering violation "
15400:                 f"for key '{key}': element type '{element_type}' has chunk "
15401:                 f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
15402:                 f"[question_id={question_id}, chunk_id={chunk_id}, "
15403:                 f"correlation_id={correlation_id}]"
15404:             )
15405:
15406:     validated_count += 1
15407:
15408: return validated_count
15409:
15410:
15411: def validate_phase6_schema_compatibility(
15412:     question: dict[str, Any],
15413:     chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
15414:     chunk_id: str,
15415:     policy_area_id: str,
15416:     correlation_id: str,
15417: ) -> int:
15418:     """Phase 6.4: Orchestrator - Coordinate complete validation pipeline.
15419:
15420:     Invokes structural then semantic layers in sequence, captures element count
15421:     return value, emits debug log with has_required_fields and has_minimum_thresholds
15422:     computed via any-element-iteration, logs info warning for None chunk schema
15423:     with non-None question schema, and allows TypeError-ValueError propagation
15424:     to outer handler without try-except wrapping.
15425:
15426:     This is the main entry point for Phase 6 validation, designed to integrate
15427:     into the build_with_chunk_matrix loop after Phase 5 (signal resolution) and
15428:     before construct_task.
15429:
15430:     Args:
15431:         question: Question dictionary from questionnaire
15432:         chunk_expected_elements: expected_elements from chunk routing result
15433:         chunk_id: Chunk identifier for logging
15434:         policy_area_id: Policy area identifier for task ID construction
15435:         correlation_id: Correlation ID for distributed tracing
15436:
15437:     Returns:
15438:         Validated element count (number of elements validated)
15439:
15440:     Raises:
15441:         TypeError: If either schema has invalid type (propagated from Phase 6.2)
15442:         ValueError: If validation fails (propagated from Phase 6.1, 6.2, or 6.3)
15443:
15444:     Integration Point:
15445:         Called within build_with_chunk_matrix loop:
15446:             1. After Phase 5: Signal resolution completes
15447:             2. Before construct_task: Task construction begins
15448:             3. No try-except wrapper: Exceptions propagate to outer handler
```

```
15449:  
15450:     Orchestration Flow:  
15451:         Phase 6.1: Extract and classify schemas  
15452:         Phase 6.2: Validate structural compatibility  
15453:         Phase 6.3: Validate semantic constraints (if both schemas non-None)  
15454:         Phase 6.4: Emit debug logs and return validated count  
15455:     """  
15456:     question_id = question.get("question_id", "UNKNOWN")  
15457:  
15458:     # Phase 6.1: Classification & Extraction  
15459:     (  
15460:         question_global,  
15461:         question_schema,  
15462:         chunk_schema,  
15463:         question_type,  
15464:         chunk_type,  
15465:     ) = _extract_and_classify_schemas(question, chunk_expected_elements, question_id)  
15466:  
15467:     # Construct provisional task ID for error reporting  
15468:     provisional_task_id = f"MQC-{question_global:03d}_{policy_area_id}"  
15469:  
15470:     # Phase 6.2: Structural Validation  
15471:     _validate_structural_compatibility(  
15472:         question_schema,  
15473:         chunk_schema,  
15474:         question_type,  
15475:         chunk_type,  
15476:         question_id,  
15477:         correlation_id,  
15478:     )  
15479:  
15480:     # Phase 6.3: Semantic Validation (if both schemas non-None)  
15481:     validated_count = 0  
15482:     if question_schema is not None and chunk_schema is not None:  
15483:         validated_count = _validate_semantic_constraints(  
15484:             question_schema,  
15485:             chunk_schema,  
15486:             question_type,  
15487:             chunk_type,  
15488:             provisional_task_id,  
15489:             question_id,  
15490:             chunk_id,  
15491:             correlation_id,  
15492:         )  
15493:  
15494:     # Phase 6.4: Emit debug log with has_required_fields and has_minimum_thresholds  
15495:     # Compute via any-element-iteration  
15496:     has_required_fields = False  
15497:     has_minimum_thresholds = False  
15498:  
15499:     if question_schema is not None:  
15500:         if isinstance(question_schema, list):  
15501:             has_required_fields = any(  
15502:                 elem.get("required", False)  
15503:                 for elem in question_schema  
15504:                 if isinstance(elem, dict)
```

```

15505:         )
15506:         has_minimum_thresholds = any(
15507:             "minimum" in elem for elem in question_schema if isinstance(elem, dict)
15508:         )
15509:     elif isinstance(question_schema, dict):
15510:         has_required_fields = any(
15511:             elem.get("required", False)
15512:             for elem in question_schema.values()
15513:             if isinstance(elem, dict)
15514:         )
15515:         has_minimum_thresholds = any(
15516:             "minimum" in elem
15517:                 for elem in question_schema.values()
15518:                 if isinstance(elem, dict)
15519:         )
15520:
15521:     logger.debug(
15522:         f"Phase 6 validation complete: question_id={question_id}, "
15523:         f"chunk_id={chunk_id}, provisional_task_id={provisional_task_id}, "
15524:         f"validated_count={validated_count}, "
15525:         f"has_required_fields={has_required_fields}, "
15526:         f"has_minimum_thresholds={has_minimum_thresholds}, "
15527:         f"question_type={question_type}, chunk_type={chunk_type}, "
15528:         f"correlation_id={correlation_id}"
15529:     )
15530:
15531: # Log info warning for None chunk schema with non-None question schema
15532: if question_schema is not None and chunk_schema is None:
15533:     logger.info(
15534:         f"Schema asymmetry detected: question_id={question_id}, "
15535:         f"chunk_id={chunk_id}, question_schema_type={question_type}, "
15536:         f"chunk_schema_type=none, message='Question specifies required elements "
15537:         f"but chunk provides no schema', "
15538:         f"validation_status='compatible_via_constraint_relaxation', "
15539:         f"correlation_id={correlation_id}"
15540:     )
15541:
15542: return validated_count
15543:
15544:
15545: __all__ = [
15546:     "validate_phase6_schema_compatibility",
15547:     "_extract_and_classify_schemas",
15548:     "_validate_structural_compatibility",
15549:     "_validate_semantic_constraints",
15550:     "_classify_expected_elements_type",
15551: ]
15552:
15553:
15554:
15555: =====
15556: FILE: src/farfan_pipeline/core/orchestrator/precision_tracking.py
15557: =====
15558:
15559: """
15560: Precision Improvement Tracking for Context Filtering

```

```
15561: =====
15562:
15563: Enhanced validation and comprehensive stats tracking for the 60% precision
15564: improvement target from filter_patterns_by_context integration.
15565:
15566: This module provides:
15567: 1. Enhanced get_patterns_for_context() wrapper with validation
15568: 2. Detailed validation status tracking
15569: 3. Comprehensive logging and metrics
15570: 4. Target achievement verification
15571:
15572: Usage:
15573:     >>> from farfan_pipeline.core.orchestrator.precision_tracking import (
15574:         ...     get_patterns_with_validation
15575:         ... )
15576:     >>> patterns, stats = get_patterns_with_validation(
15577:         ...     enriched_pack, document_context
15578:         ... )
15579:     >>> assert stats['integration_validated']
15580:     >>> assert stats['target_achieved']
15581:
15582: Author: F.A.R.F.A.N Pipeline
15583: Date: 2025-12-03
15584: """
15585:
15586: from datetime import datetime, timezone
15587: from typing import Any
15588:
15589: try:
15590:     import structlog
15591:
15592:     logger = structlog.get_logger(__name__)
15593: except ImportError:
15594:     import logging
15595:
15596:     logger = logging.getLogger(__name__)
15597:
15598:
15599: PRECISION_TARGET_THRESHOLD = 0.55
15600:
15601:
15602: def get_patterns_with_validation(
15603:     enriched_pack: Any,
15604:     document_context: dict[str, Any],
15605:     track_precision_improvement: bool = True,
15606: ) -> tuple[list[dict[str, Any]], dict[str, Any]]:
15607:     """
15608:         Enhanced wrapper for get_patterns_for_context() with comprehensive validation.
15609:
15610:         This function wraps EnrichedSignalPack.get_patterns_for_context() and adds:
15611:             - Pre-filtering validation
15612:             - Post-filtering verification
15613:             - Integration status checking
15614:             - Target achievement tracking
15615:             - Detailed logging
15616:
```

```
15617:     Args:
15618:         enriched_pack: EnrichedSignalPack instance
15619:         document_context: Document context dict
15620:         track_precision_improvement: Enable precision tracking
15621:
15622:     Returns:
15623:         Tuple of (filtered_patterns, comprehensive_stats) with enhanced fields:
15624:             - validation_timestamp: ISO timestamp
15625:             - validation_details: Detailed validation info
15626:             - target_achieved: Boolean for 60% target
15627:             - validation_status: Status string
15628:             - target_status: Status string
15629:             - pre_filter_count: Patterns before filtering
15630:             - post_filter_count: Patterns after filtering
15631:             - filtering_successful: Boolean validation
15632:
15633:     Example:
15634:         >>> enriched = create_enriched_signal_pack(base_pack)
15635:         >>> context = create_document_context(section='budget', chapter=3)
15636:         >>> patterns, stats = get_patterns_with_validation(enriched, context)
15637:         >>> print(f"Validation: {stats['validation_status']}")
15638:         >>> print(f"Target: {stats['target_status']}")
15639:         >>> assert stats['integration_validated']
15640:         >>> assert stats['target_achieved']
15641: """
15642: if not isinstance(document_context, dict):
15643:     logger.warning(
15644:         "invalid_document_context_type",
15645:         context_type=type(document_context).__name__,
15646:         expected="dict",
15647:     )
15648:     document_context = {}
15649:
15650: validation_timestamp = datetime.now(timezone.utc).isoformat()
15651:
15652: pre_filter_count = (
15653:     len(enriched_pack.patterns) if hasattr(enriched_pack, "patterns") else 0
15654: )
15655:
15656: filtered, base_stats = enriched_pack.get_patterns_for_context(
15657:     document_context, track_precision_improvement=track_precision_improvement
15658: )
15659:
15660: post_filter_count = len(filtered)
15661:
15662: validation_details = {
15663:     "filter_function_called": True,
15664:     "pre_filter_count": pre_filter_count,
15665:     "post_filter_count": post_filter_count,
15666:     "context_fields": list(document_context.keys()),
15667:     "context_field_count": len(document_context),
15668:     "filtering_successful": post_filter_count <= pre_filter_count,
15669:     "patterns_reduced": pre_filter_count - post_filter_count,
15670:     "reduction_percentage": (
15671:         (pre_filter_count - post_filter_count) / pre_filter_count * 100
15672:         if pre_filter_count > 0

```

```
15673:         else 0.0
15674:     ),
15675: }
15676:
15677: enhanced_stats = {**base_stats}
15678: enhanced_stats["validation_timestamp"] = validation_timestamp
15679: enhanced_stats["validation_details"] = validation_details
15680: enhanced_stats["pre_filter_count"] = pre_filter_count
15681: enhanced_stats["post_filter_count"] = post_filter_count
15682: enhanced_stats["filtering_successful"] = validation_details["filtering_successful"]
15683:
15684: if track_precision_improvement:
15685:     integration_validated = base_stats.get("integration_validated", False)
15686:     false_positive_reduction = base_stats.get("false_positive_reduction", 0.0)
15687:     target_achieved = false_positive_reduction >= PRECISION_TARGET_THRESHOLD
15688:
15689:     enhanced_stats["target_achieved"] = target_achieved
15690:
15691:     if integration_validated:
15692:         enhanced_stats["validation_status"] = "VALIDATED"
15693:         validation_message = "\u234\u223 filter_patterns_by_context integration VALIDATED"
15694:     else:
15695:         enhanced_stats["validation_status"] = "NOT_VALIDATED"
15696:         validation_message = (
15697:             "\u234\u227 filter_patterns_by_context integration NOT validated"
15698:         )
15699:
15700:     target_status = "ACHIEVED" if target_achieved else "NOT_MET"
15701:     enhanced_stats["target_status"] = target_status
15702:
15703:     if not validation_details["filtering_successful"]:
15704:         logger.error(
15705:             "context_filtering_validation_failed",
15706:             pre_filter_count=pre_filter_count,
15707:             post_filter_count=post_filter_count,
15708:             reason="filtered_count_exceeds_original",
15709:         )
15710:     enhanced_stats["integration_validated"] = False
15711:     enhanced_stats["validation_status"] = "FAILED"
15712:
15713:     logger.info(
15714:         "enhanced_context_filtering_validation",
15715:         pre_filter_count=pre_filter_count,
15716:         post_filter_count=post_filter_count,
15717:         patterns_reduced=validation_details["patterns_reduced"],
15718:         reduction_percentage=f"{validation_details['reduction_percentage']:.1f}%",
15719:         filter_rate=f"{base_stats.get('filter_rate', 0.0):.1%}",
15720:         precision_improvement=f"{base_stats.get('precision_improvement', 0.0):.1%}",
15721:         false_positive_reduction=f"{false_positive_reduction:.1%}",
15722:         integration_validated=integration_validated,
15723:         validation_status=enhanced_stats["validation_status"],
15724:         target_achieved=target_achieved,
15725:         target_status=target_status,
15726:         validation_message=validation_message,
15727:         validation_timestamp=validation_timestamp,
15728:     )
```

```
15729:  
15730:         if target_achieved:  
15731:             logger.info(  
15732:                 "precision_target_achieved",  
15733:                 false_positive_reduction=f"{{false_positive_reduction:.1%}}",  
15734:                 target_threshold=f"{{PRECISION_TARGET_THRESHOLD:.1%}}",  
15735:                 message="\\234\\223 60% precision improvement target ACHIEVED",  
15736:             )  
15737:         else:  
15738:             logger.warning(  
15739:                 "precision_target_not_met",  
15740:                 false_positive_reduction=f"{{false_positive_reduction:.1%}}",  
15741:                 target_threshold=f"{{PRECISION_TARGET_THRESHOLD:.1%}}",  
15742:                 shortfall=f"{{(PRECISION_TARGET_THRESHOLD - false_positive_reduction):.1%}}",  
15743:                 message="\\234\\227 60% precision improvement target NOT met",  
15744:             )  
15745:     else:  
15746:         enhanced_stats["target_achieved"] = False  
15747:         enhanced_stats["validation_status"] = "TRACKING_DISABLED"  
15748:         enhanced_stats["target_status"] = "UNKNOWN"  
15749:         logger.debug("context_filtering_applied_without_tracking", **validation_details)  
15750:  
15751:     return filtered, enhanced_stats  
15752:  
15753:  
15754: def validate_filter_integration(  
15755:     enriched_pack: Any, test_contexts: list[dict[str, Any]] | None = None  
15756: ) -> dict[str, Any]:  
15757: """  
15758:     Comprehensive validation of filter_patterns_by_context integration.  
15759:  
15760:     Tests the filtering functionality across multiple contexts and validates:  
15761:     - Integration is working correctly  
15762:     - Patterns are being filtered  
15763:     - 60% target is achievable  
15764:     - No errors occur during filtering  
15765:  
15766:     Args:  
15767:         enriched_pack: EnrichedSignalPack instance to test  
15768:         test_contexts: Optional list of test contexts. If None, uses defaults.  
15769:  
15770:     Returns:  
15771:         Validation report dict with:  
15772:             - total_tests: Number of contexts tested  
15773:             - successful_tests: Tests that completed without error  
15774:             - integration_validated: Overall integration status  
15775:             - target_achieved_count: Number of tests achieving 60% target  
15776:             - target_achievement_rate: Percentage achieving target  
15777:             - average_filter_rate: Average pattern reduction  
15778:             - average_fp_reduction: Average false positive reduction  
15779:             - validation_summary: Human-readable summary  
15780:  
15781:     Example:  
15782:         >>> enriched = create_enriched_signal_pack(base_pack)  
15783:         >>> report = validate_filter_integration(enriched)  
15784:         >>> print(report['validation_summary'])
```

```
15785:         >>> assert report['integration_validated']
15786:         >>> assert report['target_achievement_rate'] > 0.5
15787:         """
15788:         if test_contexts is None:
15789:             test_contexts = [
15790:                 {},
15791:                 {"section": "budget"},
15792:                 {"section": "indicators", "chapter": 5},
15793:                 {"section": "financial", "chapter": 2, "page": 10},
15794:                 {"policy_area": "economic_development"},
15795:             ]
15796:
15797:         results = []
15798:         errors = []
15799:
15800:         for idx, context in enumerate(test_contexts):
15801:             try:
15802:                 patterns, stats = get_patterns_with_validation(
15803:                     enriched_pack, context, track_precision_improvement=True
15804:                 )
15805:                 results.append(stats)
15806:             except Exception as e:
15807:                 logger.error(
15808:                     "filter_validation_test_failed",
15809:                     test_index=idx,
15810:                     context=context,
15811:                     error=str(e),
15812:                     error_type=type(e).__name__,
15813:                 )
15814:             errors.append(
15815:                 {
15816:                     "test_index": idx,
15817:                     "context": context,
15818:                     "error": str(e),
15819:                     "error_type": type(e).__name__,
15820:                 }
15821:             )
15822:
15823:         total_tests = len(test_contexts)
15824:         successful_tests = len(results)
15825:         failed_tests = len(errors)
15826:
15827:         if successful_tests == 0:
15828:             return {
15829:                 "total_tests": total_tests,
15830:                 "successful_tests": 0,
15831:                 "failed_tests": failed_tests,
15832:                 "integration_validated": False,
15833:                 "target_achieved_count": 0,
15834:                 "target_achievement_rate": 0.0,
15835:                 "average_filter_rate": 0.0,
15836:                 "average_fp_reduction": 0.0,
15837:                 "errors": errors,
15838:                 "validation_summary": "\u262d\u262d\u262d ALL TESTS FAILED - Integration NOT working",
15839:             }
15840:
```

```
15841:     integration_validated_count = sum(
15842:         1 for r in results if r.get("integration_validated", False)
15843:     )
15844:     target_achieved_count = sum(1 for r in results if r.get("target_achieved", False))
15845:
15846:     average_filter_rate = (
15847:         sum(r.get("filter_rate", 0.0) for r in results) / successful_tests
15848:     )
15849:     average_fp_reduction = (
15850:         sum(r.get("false_positive_reduction", 0.0) for r in results) / successful_tests
15851:     )
15852:
15853:     integration_rate = integration_validated_count / successful_tests
15854:     target_achievement_rate = target_achieved_count / successful_tests
15855:
15856:     overall_integration_validated = integration_rate >= 0.8
15857:
15858:     validation_summary = (
15859:         f"Filter Integration Validation Report:\n"
15860:         f"  Tests: {successful_tests}/{total_tests} successful ({failed_tests} failed)\n"
15861:         f"  Integration validated: {integration_validated_count}/{successful_tests} "
15862:         f"({{integration_rate:.0%}})\n"
15863:         f"  60% target achieved: {target_achieved_count}/{successful_tests} "
15864:         f"({{target_achievement_rate:.0%}})\n"
15865:         f"  Average filter rate: {average_filter_rate:.1%}\n"
15866:         f"  Average FP reduction: {average_fp_reduction:.1%}\n"
15867:         f"  Overall status: "
15868:         f"{{'VALIDATED' if overall_integration_validated else 'NOT VALIDATED'}}\n"
15869:         f"  Target status: "
15870:         f"{{'ACHIEVABLE' if target_achievement_rate > 0 else 'NOT ACHIEVABLE'}}"
15871:     )
15872:
15873:     report = {
15874:         "total_tests": total_tests,
15875:         "successful_tests": successful_tests,
15876:         "failed_tests": failed_tests,
15877:         "integration_validated": overall_integration_validated,
15878:         "integration_validated_count": integration_validated_count,
15879:         "integration_rate": integration_rate,
15880:         "target_achieved_count": target_achieved_count,
15881:         "target_achievement_rate": target_achievement_rate,
15882:         "average_filter_rate": average_filter_rate,
15883:         "average_fp_reduction": average_fp_reduction,
15884:         "max_fp_reduction": (
15885:             max(r.get("false_positive_reduction", 0.0) for r in results)
15886:             if results
15887:             else 0.0
15888:         ),
15889:         "min_fp_reduction": (
15890:             min(r.get("false_positive_reduction", 0.0) for r in results)
15891:             if results
15892:             else 0.0
15893:         ),
15894:         "errors": errors,
15895:         "validation_summary": validation_summary,
15896:         "all_results": results,
```

```
15897:     }
15898:
15899:     logger.info(
15900:         "filter_integration_validation_complete",
15901:         total_tests=total_tests,
15902:         successful_tests=successful_tests,
15903:         failed_tests=failed_tests,
15904:         integration_validated=overall_integration_validated,
15905:         target_achievement_rate=f"{target_achievement_rate:.0%}",
15906:         summary=validation_summary,
15907:     )
15908:
15909:     return report
15910:
15911:
15912: def create_precision_tracking_session(
15913:     enriched_pack: Any, session_id: str | None = None
15914: ) -> dict[str, Any]:
15915:     """
15916:         Create a precision tracking session for continuous monitoring.
15917:
15918:     This creates a session object that tracks multiple measurements over time,
15919:     useful for monitoring precision improvement during production analysis.
15920:
15921:     Args:
15922:         enriched_pack: EnrichedSignalPack instance
15923:         session_id: Optional session identifier
15924:
15925:     Returns:
15926:         Session object with tracking state and methods
15927:
15928:     Example:
15929:         >>> session = create_precision_tracking_session(enriched_pack, "prod_001")
15930:         >>> # Use session throughout analysis...
15931:         >>> results = finalize_precision_tracking_session(session)
15932:     """
15933:     from datetime import datetime, timezone
15934:     from uuid import uuid4
15935:
15936:     if session_id is None:
15937:         session_id = f"precision_session_{uuid4().hex[:8]}"
15938:
15939:     session = {
15940:         "session_id": session_id,
15941:         "start_timestamp": datetime.now(timezone.utc).isoformat(),
15942:         "enriched_pack": enriched_pack,
15943:         "measurements": [],
15944:         "measurement_count": 0,
15945:         "contexts_tested": [],
15946:         "cumulative_stats": {
15947:             "total_patterns_processed": 0,
15948:             "total_patterns_filtered": 0,
15949:             "total_filtering_time_ms": 0.0,
15950:         },
15951:         "status": "ACTIVE",
15952:     }
```

```
15953:
15954:     logger.info(
15955:         "precision_tracking_session_created",
15956:         session_id=session_id,
15957:         start_timestamp=session["start_timestamp"],
15958:     )
15959:
15960:     return session
15961:
15962:
15963: def add_measurement_to_session(
15964:     session: dict[str, Any],
15965:     document_context: dict[str, Any],
15966:     track_precision: bool = True,
15967: ) -> tuple[list[dict[str, Any]], dict[str, Any]]:
15968:     """
15969:     Add a measurement to an active precision tracking session.
15970:
15971:     Args:
15972:         session: Active session from create_precision_tracking_session
15973:         document_context: Document context for this measurement
15974:         track_precision: Enable precision tracking
15975:
15976:     Returns:
15977:         Tuple of (filtered_patterns, stats) from get_patterns_for_context
15978:
15979:     Example:
15980:         >>> session = create_precision_tracking_session(enriched_pack)
15981:         >>> for context in contexts:
15982:             ...     patterns, stats = add_measurement_to_session(session, context)
15983:         """
15984:     if session["status"] != "ACTIVE":
15985:         logger.warning(
15986:             "measurement_to_inactive_session",
15987:             session_id=session["session_id"],
15988:             status=session["status"],
15989:         )
15990:
15991:     enriched_pack = session["enriched_pack"]
15992:     patterns, stats = get_patterns_with_validation(
15993:         enriched_pack, document_context, track_precision
15994:     )
15995:
15996:     session["measurements"].append(stats)
15997:     session["measurement_count"] += 1
15998:     session["contexts_tested"].append(document_context)
15999:
16000:     session["cumulative_stats"]["total_patterns_processed"] += stats.get(
16001:         "total_patterns", 0
16002:     )
16003:     session["cumulative_stats"]["total_patterns_filtered"] += stats.get(
16004:         "total_patterns", 0
16005:     ) - stats.get("passed", 0)
16006:     session["cumulative_stats"]["total_filtering_time_ms"] += stats.get(
16007:         "filtering_duration_ms", 0.0
16008:     )
```

```
16009:  
16010:     return patterns, stats  
16011:  
16012:  
16013: def finalize_precision_tracking_session(  
16014:     session: dict[str, Any], generate_full_report: bool = True  
16015: ) -> dict[str, Any]:  
16016:     """  
16017:         Finalize a precision tracking session and generate summary.  
16018:  
16019:     Args:  
16020:         session: Active session to finalize  
16021:         generate_full_report: Include full detailed report  
16022:  
16023:     Returns:  
16024:         Finalized session report with comprehensive metrics  
16025:  
16026:     Example:  
16027:         >>> session = create_precision_tracking_session(enriched_pack)  
16028:         >>> # ... add measurements ...  
16029:         >>> results = finalize_precision_tracking_session(session)  
16030:         >>> print(results['summary'])  
16031:     """  
16032:     from datetime import datetime, timezone  
16033:  
16034:     from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (  
16035:         generate_precision_improvement_report,  
16036:     )  
16037:  
16038:     end_timestamp = datetime.now(timezone.utc).isoformat()  
16039:     session["end_timestamp"] = end_timestamp  
16040:     session["status"] = "FINALIZED"  
16041:  
16042:     if not session["measurements"]:  
16043:         return {  
16044:             "session_id": session["session_id"],  
16045:             "status": "FINALIZED",  
16046:             "measurement_count": 0,  
16047:             "summary": "No measurements recorded",  
16048:         }  
16049:  
16050:     full_report = None  
16051:     if generate_full_report:  
16052:         full_report = generate_precision_improvement_report(  
16053:             session["measurements"], include_detailed_breakdown=True  
16054:         )  
16055:  
16056:     session_summary = {  
16057:         "session_id": session["session_id"],  
16058:         "start_timestamp": session["start_timestamp"],  
16059:         "end_timestamp": end_timestamp,  
16060:         "status": session["status"],  
16061:         "measurement_count": session["measurement_count"],  
16062:         "cumulative_stats": session["cumulative_stats"],  
16063:         "contexts_tested_count": len(session["contexts_tested"]),  
16064:     }
```

```
16065:
16066:     if full_report:
16067:         session_summary["aggregate_report"] = full_report
16068:         session_summary["summary"] = full_report["summary"]
16069:         session_summary["target_achievement_rate"] = full_report[
16070:             "target_achievement_rate"
16071:         ]
16072:         session_summary["integration_validated"] = full_report["validation_rate"] >= 0.8
16073:         session_summary["validation_health"] = full_report["validation_health"]
16074:
16075:     logger.info(
16076:         "precision_tracking_session_finalized",
16077:         session_id=session["session_id"],
16078:         measurement_count=session["measurement_count"],
16079:         total_patterns_processed=session["cumulative_stats"][
16080:             "total_patterns_processed"
16081:         ],
16082:         total_filtering_time_ms=session["cumulative_stats"]["total_filtering_time_ms"],
16083:         target_achievement_rate=(session_summary.get("target_achievement_rate", 0.0)),
16084:     )
16085:
16086:     return session_summary
16087:
16088:
16089: def compare_precision_across_policy_areas(
16090:     policy_area_packs: dict[str, Any], test_contexts: list[dict[str, Any]] | None = None
16091: ) -> dict[str, Any]:
16092: """
16093:     Compare precision improvement across multiple policy areas.
16094:
16095:     Useful for identifying which policy areas achieve the 60% target and which need improvement.
16096:
16097:     Args:
16098:         policy_area_packs: Dict mapping policy_area_id to EnrichedSignalPack
16099:         test_contexts: Optional test contexts (uses defaults if None)
16100:
16101:     Returns:
16102:         Comparison report with per-area metrics and rankings
16103:
16104:     Example:
16105:         >>> packs = {
16106:             ...      "PA01": create_enriched_signal_pack(base_pack_01),
16107:             ...      "PA02": create_enriched_signal_pack(base_pack_02),
16108:             ...
16109:             >>> comparison = compare_precision_across_policy_areas(packs)
16110:             >>> print(comparison['rankings']['by_target_achievement'])
16111: """
16112: from farfan_pipeline.core.orchestrator.signal_intelligence_layer import (
16113:     generate_precision_improvement_report,
16114: )
16115:
16116: if test_contexts is None:
16117:     test_contexts = [
16118:         {},
16119:         {"section": "budget"}, {"section": "indicators"},
```

```
16121:         {"section": "financial"},  
16122:     ]  
16123:  
16124:     area_results = {}  
16125:  
16126:     for policy_area_id, enriched_pack in policy_area_packs.items():  
16127:         measurements = []  
16128:         for context in test_contexts:  
16129:             try:  
16130:                 _, stats = enriched_pack.get_patterns_for_context(  
16131:                     context, track_precision_improvement=True  
16132:                 )  
16133:                 measurements.append(stats)  
16134:             except Exception as e:  
16135:                 logger.error(  
16136:                     "policy_area_precision_test_failed",  
16137:                     policy_area=policy_area_id,  
16138:                     context=context,  
16139:                     error=str(e),  
16140:                 )  
16141:  
16142:             if measurements:  
16143:                 report = generate_precision_improvement_report(  
16144:                     measurements, include_detailed_breakdown=False  
16145:                 )  
16146:                 area_results[policy_area_id] = report  
16147:  
16148:     if not area_results:  
16149:         return {  
16150:             "policy_areas_tested": 0,  
16151:             "comparison_status": "FAILED",  
16152:             "message": "No successful measurements",  
16153:         }  
16154:  
16155:     rankings = {  
16156:         "by_target_achievement": sorted(  
16157:             area_results.items(),  
16158:             key=lambda x: x[1]["target_achievement_rate"],  
16159:             reverse=True,  
16160:         ),  
16161:         "by_avg_fp_reduction": sorted(  
16162:             area_results.items(),  
16163:             key=lambda x: x[1]["avg_false_positive_reduction"],  
16164:             reverse=True,  
16165:         ),  
16166:         "by_validation_rate": sorted(  
16167:             area_results.items(), key=lambda x: x[1]["validation_rate"], reverse=True  
16168:         ),  
16169:     }  
16170:  
16171:     best_performer = rankings["by_target_achievement"][0]  
16172:     worst_performer = rankings["by_target_achievement"][-1]  
16173:  
16174:     areas_meeting_target = sum(  
16175:         1  
16176:         for _, report in area_results.items()
```

```
16177:         if report["max_false_positive_reduction"] >= PRECISION_TARGET_THRESHOLD
16178:     )
16179:
16180:     comparison_summary = (
16181:         f"Policy Area Precision Comparison:\n"
16182:         f"  Areas tested: {len(area_results)}\n"
16183:         f"  Areas meeting 60% target: {areas_meeting_target}/{len(area_results)}\n"
16184:         f"  Best performer: {best_performer[0]} "
16185:         f"({100*best_performer[1]['target_achievement_rate']):.0f}% target achievement)\n"
16186:         f"  Worst performer: {worst_performer[0]} "
16187:         f"({100*worst_performer[1]['target_achievement_rate']):.0f}% target achievement)\n"
16188:         f"  Overall status: "
16189:         f"'{â\234\223 GOOD' if areas_meeting_target >= len(area_results) * 0.7 else 'â\234\227 NEEDS IMPROVEMENT'}"
16190:     )
16191:
16192:     return {
16193:         "policy_areas_tested": len(area_results),
16194:         "areas_meeting_target": areas_meeting_target,
16195:         "target_achievement_coverage": areas_meeting_target / len(area_results),
16196:         "rankings": rankings,
16197:         "best_performer": {
16198:             "policy_area": best_performer[0],
16199:             "metrics": best_performer[1],
16200:         },
16201:         "worst_performer": {
16202:             "policy_area": worst_performer[0],
16203:             "metrics": worst_performer[1],
16204:         },
16205:         "all_results": area_results,
16206:         "comparison_summary": comparison_summary,
16207:     }
16208:
16209:
16210: def export_precision_metrics_for_monitoring(
16211:     measurements: list[dict[str, Any]], output_format: str = "json"
16212: ) -> str | dict[str, Any]:
16213: """
16214:     Export precision metrics in format suitable for external monitoring systems.
16215:
16216:     Args:
16217:         measurements: List of stats dicts from get_patterns_for_context
16218:         output_format: 'json', 'prometheus', or 'datadog'
16219:
16220:     Returns:
16221:         Formatted metrics string or dict
16222:
16223:     Example:
16224:         >>> measurements = [...]
16225:         >>> metrics = export_precision_metrics_for_monitoring(measurements, 'json')
16226: """
16227: import json
16228: from datetime import datetime, timezone
16229:
16230: timestamp = datetime.now(timezone.utc).isoformat()
16231:
16232: if not measurements:
```

```
16233:         if output_format == "json":
16234:             return json.dumps({"error": "No measurements", "timestamp": timestamp})
16235:         return ""
16236:
16237:     total = len(measurements)
16238:     meets_target = sum(
16239:         1
16240:         for m in measurements
16241:             if m.get("false_positive_reduction", 0.0) >= PRECISION_TARGET_THRESHOLD
16242:     )
16243:     validated = sum(1 for m in measurements if m.get("integration_validated", False))
16244:
16245:     avg_fp_reduction = (
16246:         sum(m.get("false_positive_reduction", 0.0) for m in measurements) / total
16247:     )
16248:     avg_filter_rate = sum(m.get("filter_rate", 0.0) for m in measurements) / total
16249:
16250:     if output_format == "json":
16251:         return json.dumps(
16252:             {
16253:                 "timestamp": timestamp,
16254:                 "measurement_count": total,
16255:                 "target_achievement_count": meets_target,
16256:                 "target_achievement_rate": meets_target / total,
16257:                 "integration_validated_count": validated,
16258:                 "integration_validation_rate": validated / total,
16259:                 "avg_false_positive_reduction": avg_fp_reduction,
16260:                 "avg_filter_rate": avg_filter_rate,
16261:                 "meets_60_percent_target": meets_target / total >= 0.5,
16262:             },
16263:             indent=2,
16264:         )
16265:
16266:     elif output_format == "prometheus":
16267:         lines = [
16268:             "# HELP precision_target_achievement_rate Rate of measurements meeting 60% target",
16269:             "# TYPE precision_target_achievement_rate gauge",
16270:             f"precision_target_achievement_rate {meets_target / total}",
16271:             "# HELP precision_avg_fp_reduction Average false positive reduction",
16272:             "# TYPE precision_avg_fp_reduction gauge",
16273:             f"precision_avg_fp_reduction {avg_fp_reduction}",
16274:             "# HELP precision_measurement_count Total measurements",
16275:             "# TYPE precision_measurement_count counter",
16276:             f"precision_measurement_count {total}",
16277:         ]
16278:         return "\n".join(lines)
16279:
16280:     elif output_format == "datadog":
16281:         return json.dumps(
16282:             [
16283:                 {
16284:                     "metric": "farfan.precision.target_achievement_rate",
16285:                     "points": [
16286:                         [
16287:                             int(datetime.now(timezone.utc).timestamp()),
16288:                             meets_target / total,
```

```
16289:         ]
16290:     ],
16291:     "type": "gauge",
16292:     "tags": ["component:context_filtering"],
16293:   },
16294:   {
16295:     "metric": "farfan.precision.avg_fp_reduction",
16296:     "points": [
16297:       [int(datetime.now(timezone.utc).timestamp()), avg_fp_reduction]
16298:     ],
16299:     "type": "gauge",
16300:     "tags": ["component:context_filtering"],
16301:   },
16302:   {
16303:     "metric": "farfan.precision.measurement_count",
16304:     "points": [[int(datetime.now(timezone.utc).timestamp()), total]],
16305:     "type": "count",
16306:     "tags": ["component:context_filtering"],
16307:   },
16308: ],
16309:   indent=2,
16310: )
16311:
16312: return ""
16313:
16314:
16315: __all__ = [
16316:     "get_patterns_with_validation",
16317:     "validate_filter_integration",
16318:     "create_precision_tracking_session",
16319:     "add_measurement_to_session",
16320:     "finalize_precision_tracking_session",
16321:     "compare_precision_across_policy_areas",
16322:     "export_precision_metrics_for_monitoring",
16323:     "PRECISION_TARGET_THRESHOLD",
16324: ]
16325:
16326:
16327:
16328: =====
16329: FILE: src/farfan_pipeline/core/orchestrator/resource_alerts.py
16330: =====
16331:
16332: """Resource Pressure Alerting and Observability.
16333:
16334: Provides comprehensive alerting and monitoring for resource management:
16335: - Structured logging for resource events
16336: - Alert thresholds and notifications
16337: - Integration with external monitoring systems
16338: - Historical trend analysis
16339: """
16340:
16341: from __future__ import annotations
16342:
16343: import json
16344: import logging
```

```
16345: from collections import defaultdict
16346: from datetime import datetime, timedelta
16347: from enum import Enum
16348: from typing import Any, Callable
16349:
16350: from farfan_pipeline.core.orchestrator.resource_manager import (
16351:     ResourcePressureEvent,
16352:     ResourcePressureLevel,
16353: )
16354:
16355: logger = logging.getLogger(__name__)
16356:
16357:
16358: class AlertSeverity(Enum):
16359:     """Alert severity levels."""
16360:
16361:     INFO = "info"
16362:     WARNING = "warning"
16363:     ERROR = "error"
16364:     CRITICAL = "critical"
16365:
16366:
16367: class AlertChannel(Enum):
16368:     """Alert delivery channels."""
16369:
16370:     LOG = "log"
16371:     WEBHOOK = "webhook"
16372:     SIGNAL = "signal"
16373:     STDOUT = "stdout"
16374:
16375:
16376: class ResourceAlert:
16377:     """Individual resource alert."""
16378:
16379:     def __init__(
16380:         self,
16381:         severity: AlertSeverity,
16382:         title: str,
16383:         message: str,
16384:         event: ResourcePressureEvent,
16385:         metadata: dict[str, Any] | None = None,
16386:     ) -> None:
16387:         self.severity = severity
16388:         self.title = title
16389:         self.message = message
16390:         self.event = event
16391:         self.metadata = metadata or {}
16392:         self.timestamp = datetime.utcnow()
16393:         self.alert_id = f"alert_{self.timestamp.isoformat()}_{id(self)}"
16394:
16395:     def to_dict(self) -> dict[str, Any]:
16396:         """Convert alert to dictionary."""
16397:         return {
16398:             "alert_id": self.alert_id,
16399:             "timestamp": self.timestamp.isoformat(),
16400:             "severity": self.severity.value,
```

```
16401:         "title": self.title,
16402:         "message": self.message,
16403:         "event": {
16404:             "timestamp": self.event.timestamp.isoformat(),
16405:             "pressure_level": self.event.pressure_level.value,
16406:             "cpu_percent": self.event.cpu_percent,
16407:             "memory_mb": self.event.memory_mb,
16408:             "memory_percent": self.event.memory_percent,
16409:             "worker_count": self.event.worker_count,
16410:             "active_executors": self.event.active_executors,
16411:             "degradation_applied": self.event.degradation_applied,
16412:             "circuit_breakers_open": self.event.circuit_breakers_open,
16413:         },
16414:         "metadata": self.metadata,
16415:     }
16416:
16417:     def to_json(self) -> str:
16418:         """Convert alert to JSON string."""
16419:         return json.dumps(self.to_dict(), indent=2)
16420:
16421:
16422: class AlertThresholds:
16423:     """Configurable alert thresholds."""
16424:
16425:     def __init__(
16426:         self,
16427:         memory_warning_percent: float = 75.0,
16428:         memory_critical_percent: float = 85.0,
16429:         cpu_warning_percent: float = 75.0,
16430:         cpu_critical_percent: float = 85.0,
16431:         circuit_breaker_warning_count: int = 3,
16432:         degradation_critical_count: int = 3,
16433:     ) -> None:
16434:         self.memory_warning_percent = memory_warning_percent
16435:         self.memory_critical_percent = memory_critical_percent
16436:         self.cpu_warning_percent = cpu_warning_percent
16437:         self.cpu_critical_percent = cpu_critical_percent
16438:         self.circuit_breaker_warning_count = circuit_breaker_warning_count
16439:         self.degradation_critical_count = degradation_critical_count
16440:
16441:
16442: class ResourceAlertManager:
16443:     """Manages resource pressure alerts and notifications."""
16444:
16445:     def __init__(
16446:         self,
16447:         thresholds: AlertThresholds | None = None,
16448:         channels: list[AlertChannel] | None = None,
16449:         webhook_url: str | None = None,
16450:         signal_callback: Callable[[ResourceAlert], None] | None = None,
16451:     ) -> None:
16452:         self.thresholds = thresholds or AlertThresholds()
16453:         self.channels = channels or [AlertChannel.LOG]
16454:         self.webhook_url = webhook_url
16455:         self.signal_callback = signal_callback
16456:
```

```
16457:     self.alert_history: list[ResourceAlert] = []
16458:     self.alert_counts: dict[str, int] = defaultdict(int)
16459:     self.suppressed_alerts: set[str] = set()
16460:     self.last_alert_times: dict[str, datetime] = {}
16461:
16462:     def process_event(self, event: ResourcePressureEvent) -> list[ResourceAlert]:
16463:         """Process resource pressure event and generate alerts."""
16464:         alerts: list[ResourceAlert] = []
16465:
16466:         memory_alert = self._check_memory_threshold(event)
16467:         if memory_alert:
16468:             alerts.append(memory_alert)
16469:
16470:         cpu_alert = self._check_cpu_threshold(event)
16471:         if cpu_alert:
16472:             alerts.append(cpu_alert)
16473:
16474:         pressure_alert = self._check_pressure_level(event)
16475:         if pressure_alert:
16476:             alerts.append(pressure_alert)
16477:
16478:         circuit_breaker_alert = self._check_circuit_breakers(event)
16479:         if circuit_breaker_alert:
16480:             alerts.append(circuit_breaker_alert)
16481:
16482:         degradation_alert = self._check_degradation(event)
16483:         if degradation_alert:
16484:             alerts.append(degradation_alert)
16485:
16486:         for alert in alerts:
16487:             self._dispatch_alert(alert)
16488:             self.alert_history.append(alert)
16489:             self.alert_counts[alert.severity.value] += 1
16490:
16491:     return alerts
16492:
16493:     def _check_memory_threshold(
16494:         self, event: ResourcePressureEvent
16495:     ) -> ResourceAlert | None:
16496:         """Check if memory usage exceeds thresholds."""
16497:         if event.memory_percent >= self.thresholds.memory_critical_percent:
16498:             return ResourceAlert(
16499:                 severity=AlertSeverity.CRITICAL,
16500:                 title="Critical Memory Usage",
16501:                 message=f"Memory usage at {event.memory_percent:.1f}% "
16502:                 f"({event.memory_mb:.1f} MB)",
16503:                 event=event,
16504:                 metadata={"threshold": self.thresholds.memory_critical_percent},
16505:             )
16506:
16507:         if event.memory_percent >= self.thresholds.memory_warning_percent:
16508:             if self._should_alert("memory_warning", minutes=5):
16509:                 return ResourceAlert(
16510:                     severity=AlertSeverity.WARNING,
16511:                     title="High Memory Usage",
16512:                     message=f"Memory usage at {event.memory_percent:.1f}% "
```

```
16513:             f"({event.memory_mb:.1f} MB)",
16514:             event=event,
16515:             metadata={"threshold": self.thresholds.memory_warning_percent},
16516:         )
16517:
16518:     return None
16519:
16520: def _check_cpu_threshold(
16521:     self, event: ResourcePressureEvent
16522: ) -> ResourceAlert | None:
16523:     """Check if CPU usage exceeds thresholds."""
16524:     if event.cpu_percent >= self.thresholds.cpu_critical_percent:
16525:         return ResourceAlert(
16526:             severity=AlertSeverity.CRITICAL,
16527:             title="Critical CPU Usage",
16528:             message=f"CPU usage at {event.cpu_percent:.1f}%",
16529:             event=event,
16530:             metadata={"threshold": self.thresholds.cpu_critical_percent},
16531:         )
16532:
16533:     if event.cpu_percent >= self.thresholds.cpu_warning_percent:
16534:         if self._should_alert("cpu_warning", minutes=5):
16535:             return ResourceAlert(
16536:                 severity=AlertSeverity.WARNING,
16537:                 title="High CPU Usage",
16538:                 message=f"CPU usage at {event.cpu_percent:.1f}%",
16539:                 event=event,
16540:                 metadata={"threshold": self.thresholds.cpu_warning_percent},
16541:             )
16542:
16543:     return None
16544:
16545: def _check_pressure_level(
16546:     self, event: ResourcePressureEvent
16547: ) -> ResourceAlert | None:
16548:     """Check if pressure level warrants alert."""
16549:     if event.pressure_level == ResourcePressureLevel.EMERGENCY:
16550:         return ResourceAlert(
16551:             severity=AlertSeverity.CRITICAL,
16552:             title="Emergency Resource Pressure",
16553:             message="System under emergency resource pressure",
16554:             event=event,
16555:         )
16556:
16557:     if event.pressure_level == ResourcePressureLevel.CRITICAL:
16558:         if self._should_alert("pressure_critical", minutes=2):
16559:             return ResourceAlert(
16560:                 severity=AlertSeverity.ERROR,
16561:                 title="Critical Resource Pressure",
16562:                 message="System under critical resource pressure",
16563:                 event=event,
16564:             )
16565:
16566:     if event.pressure_level == ResourcePressureLevel.HIGH:
16567:         if self._should_alert("pressure_high", minutes=10):
16568:             return ResourceAlert(
```

```
16569:             severity=AlertSeverity.WARNING,
16570:             title="High Resource Pressure",
16571:             message="System experiencing high resource pressure",
16572:             event=event,
16573:         )
16574:
16575:     return None
16576:
16577:     def _check_circuit_breakers(
16578:         self, event: ResourcePressureEvent
16579:     ) -> ResourceAlert | None:
16580:         """Check if circuit breakers warrant alert."""
16581:         open_count = len(event.circuit_breakers_open)
16582:
16583:         if open_count >= self.thresholds.circuit_breaker_warning_count:
16584:             return ResourceAlert(
16585:                 severity=AlertSeverity.ERROR,
16586:                 title="Multiple Circuit Breakers Open",
16587:                 message=f"{open_count} circuit breakers are open: "
16588:                 f"{' , '.join(event.circuit_breakers_open)}",
16589:                 event=event,
16590:                 metadata={
16591:                     "open_count": open_count,
16592:                     "executors": event.circuit_breakers_open,
16593:                 },
16594:             )
16595:
16596:         if open_count > 0:
16597:             if self._should_alert("circuit_breaker", minutes=5):
16598:                 return ResourceAlert(
16599:                     severity=AlertSeverity.WARNING,
16600:                     title="Circuit Breaker Opened",
16601:                     message=f"Circuit breakers open for: "
16602:                     f"{' , '.join(event.circuit_breakers_open)}",
16603:                     event=event,
16604:                     metadata={"executors": event.circuit_breakers_open},
16605:                 )
16606:
16607:         return None
16608:
16609:     def _check_degradation(
16610:         self, event: ResourcePressureEvent
16611:     ) -> ResourceAlert | None:
16612:         """Check if degradation strategies warrant alert."""
16613:         degradation_count = len(event.degradation_applied)
16614:
16615:         if degradation_count >= self.thresholds.degradation_critical_count:
16616:             return ResourceAlert(
16617:                 severity=AlertSeverity.ERROR,
16618:                 title="Multiple Degradation Strategies Active",
16619:                 message=f"{degradation_count} degradation strategies applied: "
16620:                 f"{' , '.join(event.degradation_applied)}",
16621:                 event=event,
16622:                 metadata={
16623:                     "count": degradation_count,
16624:                     "strategies": event.degradation_applied,
```

```
16625:         },
16626:     )
16627:
16628:     if degradation_count > 0:
16629:         if self._should_alert("degradation", minutes=10):
16630:             return ResourceAlert(
16631:                 severity=AlertSeverity.INFO,
16632:                 title="Degradation Strategies Active",
16633:                 message=f"Active degradation: "
16634:                 f"{' , '.join(event.degradation_applied)}",
16635:                 event=event,
16636:                 metadata={"strategies": event.degradation_applied},
16637:             )
16638:
16639:     return None
16640:
16641: def _should_alert(self, alert_type: str, minutes: int = 5) -> bool:
16642:     """Check if alert should be sent (with rate limiting)."""
16643:     now = datetime.utcnow()
16644:     last_time = self.last_alert_times.get(alert_type)
16645:
16646:     if not last_time:
16647:         self.last_alert_times[alert_type] = now
16648:         return True
16649:
16650:     elapsed = (now - last_time).total_seconds() / 60
16651:     if elapsed >= minutes:
16652:         self.last_alert_times[alert_type] = now
16653:         return True
16654:
16655:     return False
16656:
16657: def _dispatch_alert(self, alert: ResourceAlert) -> None:
16658:     """Dispatch alert to configured channels."""
16659:     for channel in self.channels:
16660:         try:
16661:             if channel == AlertChannel.LOG:
16662:                 self._log_alert(alert)
16663:             elif channel == AlertChannel.WEBHOOK:
16664:                 self._send_webhook(alert)
16665:             elif channel == AlertChannel.SIGNAL:
16666:                 self._send_signal(alert)
16667:             elif channel == AlertChannel.STDOUT:
16668:                 self._print_alert(alert)
16669:         except Exception as exc:
16670:             logger.error(
16671:                 f"Failed to dispatch alert to {channel.value}: {exc}"
16672:             )
16673:
16674: def _log_alert(self, alert: ResourceAlert) -> None:
16675:     """Log alert with appropriate severity."""
16676:     extra = {
16677:         "alert_id": alert.alert_id,
16678:         "alert_severity": alert.severity.value,
16679:         "pressure_level": alert.event.pressure_level.value,
16680:         "cpu_percent": alert.event.cpu_percent,
```

```
16681:         "memory_mb": alert.event.memory_mb,
16682:     }
16683:
16684:     if alert.severity == AlertSeverity.CRITICAL:
16685:         logger.critical(f"{alert.title}: {alert.message}", extra=extra)
16686:     elif alert.severity == AlertSeverity.ERROR:
16687:         logger.error(f"{alert.title}: {alert.message}", extra=extra)
16688:     elif alert.severity == AlertSeverity.WARNING:
16689:         logger.warning(f"{alert.title}: {alert.message}", extra=extra)
16690:     else:
16691:         logger.info(f"{alert.title}: {alert.message}", extra=extra)
16692:
16693: def _send_webhook(self, alert: ResourceAlert) -> None:
16694:     """Send alert via webhook."""
16695:     if not self.webhook_url:
16696:         return
16697:
16698:     try:
16699:         import requests
16700:
16701:         requests.post(
16702:             self.webhook_url,
16703:             json=alert.to_dict(),
16704:             timeout=5,
16705:         )
16706:     except Exception as exc:
16707:         logger.error(f"Webhook alert failed: {exc}")
16708:
16709: def _send_signal(self, alert: ResourceAlert) -> None:
16710:     """Send alert via signal callback."""
16711:     if not self.signal_callback:
16712:         return
16713:
16714:     try:
16715:         self.signal_callback(alert)
16716:     except Exception as exc:
16717:         logger.error(f"Signal callback failed: {exc}")
16718:
16719: def _print_alert(self, alert: ResourceAlert) -> None:
16720:     """Print alert to stdout."""
16721:     severity_colors = {
16722:         AlertSeverity.INFO: "\033[94m",
16723:         AlertSeverity.WARNING: "\033[93m",
16724:         AlertSeverity.ERROR: "\033[91m",
16725:         AlertSeverity.CRITICAL: "\033[95m",
16726:     }
16727:     reset = "\033[0m"
16728:
16729:     color = severity_colors.get(alert.severity, reset)
16730:     print(
16731:         f"\u001b[{color}]{alert.severity.value.upper()}] {alert.title}: "
16732:         f"\u001b[{alert.message}{reset}"
16733:     )
16734:
16735: def get_alert_summary(self) -> dict[str, Any]:
16736:     """Get summary of alert history."""
```

```
16737:         now = datetime.utcnow()
16738:         hour_ago = now - timedelta(hours=1)
16739:         day_ago = now - timedelta(days=1)
16740:
16741:         recent_alerts = [
16742:             alert for alert in self.alert_history if alert.timestamp >= hour_ago
16743:         ]
16744:
16745:         daily_alerts = [
16746:             alert for alert in self.alert_history if alert.timestamp >= day_ago
16747:         ]
16748:
16749:         return {
16750:             "total_alerts": len(self.alert_history),
16751:             "last_hour": len(recent_alerts),
16752:             "last_24_hours": len(daily_alerts),
16753:             "by_severity": dict(self.alert_counts),
16754:             "recent_alerts": [alert.to_dict() for alert in recent_alerts[-10:]],
16755:         }
16756:
16757:     def clear_history(self) -> None:
16758:         """Clear alert history."""
16759:         self.alert_history.clear()
16760:         self.alert_counts.clear()
16761:         self.last_alert_times.clear()
16762:
16763:
16764:
16765: =====
16766: FILE: src/farfan_pipeline/core/orchestrator/resource_aware_executor.py
16767: =====
16768:
16769: """Resource-Aware Executor Wrapper.
16770:
16771: Integrates AdaptiveResourceManager with MethodExecutor to provide:
16772: - Automatic resource allocation before execution
16773: - Circuit breaker checks before execution
16774: - Degradation configuration injection
16775: - Execution metrics tracking
16776: - Memory and timing instrumentation
16777: """
16778:
16779: from __future__ import annotations
16780:
16781: import asyncio
16782: import logging
16783: import time
16784: from typing import TYPE_CHECKING, Any
16785:
16786: if TYPE_CHECKING:
16787:     from farfan_pipeline.core.orchestrator.core import MethodExecutor
16788:     from farfan_pipeline.core.orchestrator.resource_manager import AdaptiveResourceManager
16789:
16790: logger = logging.getLogger(__name__)
16791:
16792:
```

```
16793: class ResourceAwareExecutor:
16794:     """Wraps MethodExecutor with adaptive resource management."""
16795:
16796:     def __init__(
16797:         self,
16798:         method_executor: MethodExecutor,
16799:         resource_manager: AdaptiveResourceManager,
16800:     ) -> None:
16801:         self.method_executor = method_executor
16802:         self.resource_manager = resource_manager
16803:
16804:     async def execute_with_resource_management(
16805:         self,
16806:         executor_id: str,
16807:         context: dict[str, Any],
16808:         **kwargs: Any,
16809:     ) -> dict[str, Any]:
16810:         """Execute with full resource management integration.
16811:
16812:         Args:
16813:             executor_id: Executor identifier (e.g., "D3-Q3")
16814:             context: Execution context
16815:             **kwargs: Additional arguments for execution
16816:
16817:         Returns:
16818:             Execution result with resource metadata
16819:
16820:         Raises:
16821:             RuntimeError: If circuit breaker is open or execution fails
16822:         """
16823:         can_execute, reason = self.resource_manager.can_execute(executor_id)
16824:         if not can_execute:
16825:             logger.warning(
16826:                 f"Executor {executor_id} blocked by circuit breaker: {reason}"
16827:             )
16828:             raise RuntimeError(
16829:                 f"Executor {executor_id} unavailable: {reason}"
16830:             )
16831:
16832:         allocation = await self.resource_manager.start_executor_execution(
16833:             executor_id
16834:         )
16835:
16836:         degradation_config = allocation["degradation"]
16837:         enriched_context = self._apply_degradation(context, degradation_config)
16838:
16839:         logger.info(
16840:             f"Executing {executor_id} with resource allocation",
16841:             extra={
16842:                 "max_memory_mb": allocation["max_memory_mb"],
16843:                 "max_workers": allocation["max_workers"],
16844:                 "priority": allocation["priority"],
16845:                 "degradation_applied": degradation_config["applied_strategies"],
16846:             },
16847:         )
16848:
```

```
16849:     start_time = time.perf_counter()
16850:     success = False
16851:     result = None
16852:     error = None
16853:
16854:     try:
16855:         result = await self._execute_with_timeout(
16856:             executor_id, enriched_context, allocation, **kwargs
16857:         )
16858:         success = True
16859:         return result
16860:     except Exception as exc:
16861:         error = str(exc)
16862:         logger.error(
16863:             f"Executor {executor_id} failed: {exc}",
16864:             exc_info=True,
16865:         )
16866:         raise
16867:     finally:
16868:         duration_ms = (time.perf_counter() - start_time) * 1000
16869:
16870:         memory_mb = self._estimate_memory_usage()
16871:
16872:         await self.resource_manager.end_executor_execution(
16873:             executor_id=executor_id,
16874:             success=success,
16875:             duration_ms=duration_ms,
16876:             memory_mb=memory_mb,
16877:         )
16878:
16879:         logger.info(
16880:             f"Executor {executor_id} completed",
16881:             extra={
16882:                 "success": success,
16883:                 "duration_ms": duration_ms,
16884:                 "memory_mb": memory_mb,
16885:                 "error": error,
16886:             },
16887:         )
16888:
16889:     async def _execute_with_timeout(
16890:         self,
16891:         executor_id: str,
16892:         context: dict[str, Any],
16893:         allocation: dict[str, Any],
16894:         **kwargs: Any,
16895:     ) -> dict[str, Any]:
16896:         """Execute with timeout based on resource allocation."""
16897:         timeout_seconds = self._calculate_timeout(allocation)
16898:
16899:         try:
16900:             result = await asyncio.wait_for(
16901:                 self._execute_async(executor_id, context, **kwargs),
16902:                 timeout=timeout_seconds,
16903:             )
16904:             return result
```

```
16905:         except asyncio.TimeoutError as exc:
16906:             logger.error(
16907:                 f"Executor {executor_id} timed out after {timeout_seconds}s"
16908:             )
16909:             raise RuntimeError(
16910:                 f"Executor {executor_id} timed out"
16911:             ) from exc
16912:
16913:     async def _execute_async(
16914:         self,
16915:         executor_id: str,
16916:         context: dict[str, Any],
16917:         **kwargs: Any,
16918:     ) -> dict[str, Any]:
16919:         """Async wrapper for executor execution."""
16920:         loop = asyncio.get_event_loop()
16921:         return await loop.run_in_executor(
16922:             None, self._execute_sync, executor_id, context, kwargs
16923:         )
16924:
16925:     def _execute_sync(
16926:         self,
16927:         executor_id: str,
16928:         context: dict[str, Any],
16929:         kwargs: dict[str, Any],
16930:     ) -> dict[str, Any]:
16931:         """Synchronous execution wrapper."""
16932:         try:
16933:             from farfan_pipeline.core.orchestrator.executors import (
16934:                 D3_Q3_TraceabilityValidator,
16935:                 D4_Q2_CausalChainValidator,
16936:             )
16937:
16938:             executor_map = {
16939:                 "D3-Q3": D3_Q3_TraceabilityValidator,
16940:                 "D4-Q2": D4_Q2_CausalChainValidator,
16941:             }
16942:
16943:             executor_class = executor_map.get(executor_id)
16944:             if not executor_class:
16945:                 raise ValueError(f"Unknown executor: {executor_id}")
16946:
16947:             executor_instance = executor_class(
16948:                 executor_id=executor_id,
16949:                 config={},
16950:                 method_executor=self.method_executor,
16951:             )
16952:
16953:             return executor_instance.execute(context)
16954:
16955:         except Exception as exc:
16956:             logger.error(f"Sync execution failed: {exc}")
16957:             raise
16958:
16959:     def _apply_degradation(
16960:         self,
```

```
16961:         context: dict[str, Any],
16962:         degradation_config: dict[str, Any],
16963:     ) -> dict[str, Any]:
16964:         """Apply degradation strategies to context."""
16965:         enriched = context.copy()
16966:
16967:         enriched["_resource_constraints"] = {
16968:             "entity_limit_factor": degradation_config["entity_limit_factor"],
16969:             "disable_expensive_computations": degradation_config[
16970:                 "disable_expensive_computations"
16971:             ],
16972:             "use_simplified_methods": degradation_config["use_simplified_methods"],
16973:             "skip_optional_analysis": degradation_config["skip_optional_analysis"],
16974:             "reduce_embedding_dims": degradation_config["reduce_embedding_dims"],
16975:         }
16976:
16977:         if degradation_config["entity_limit_factor"] < 1.0:
16978:             for key in ["max_entities", "max_chunks", "max_results"]:
16979:                 if key in enriched:
16980:                     enriched[key] = int(
16981:                         enriched[key] * degradation_config["entity_limit_factor"]
16982:                     )
16983:
16984:         return enriched
16985:
16986:     def _calculate_timeout(self, allocation: dict[str, Any]) -> float:
16987:         """Calculate execution timeout based on allocation."""
16988:         base_timeout = 300.0
16989:
16990:         priority = allocation["priority"]
16991:         if priority == 1:
16992:             return base_timeout * 1.5
16993:         elif priority == 2:
16994:             return base_timeout * 1.2
16995:         else:
16996:             return base_timeout
16997:
16998:     def _estimate_memory_usage(self) -> float:
16999:         """Estimate current memory usage."""
17000:         try:
17001:             import psutil
17002:             process = psutil.Process()
17003:             return process.memory_info().rss / (1024 * 1024)
17004:         except Exception:
17005:             usage = self.resource_manager.resource_limits.get_resource_usage()
17006:             return usage.get("rss_mb", 0.0)
17007:
17008:
17009: class ResourceConstraints:
17010:     """Helper to extract and apply resource constraints in executors."""
17011:
17012:     @staticmethod
17013:     def get_constraints(context: dict[str, Any]) -> dict[str, Any]:
17014:         """Extract resource constraints from context."""
17015:         return context.get(
17016:             "_resource_constraints",
```

```
17017:         {
17018:             "entity_limit_factor": 1.0,
17019:             "disable_expensive_computations": False,
17020:             "use_simplified_methods": False,
17021:             "skip_optional_analysis": False,
17022:             "reduce_embedding_dims": False,
17023:         },
17024:     )
17025:
17026:     @staticmethod
17027:     def should_skip_expensive_computation(context: dict[str, Any]) -> bool:
17028:         """Check if expensive computations should be skipped."""
17029:         constraints = ResourceConstraints.get_constraints(context)
17030:         return constraints.get("disable_expensive_computations", False)
17031:
17032:     @staticmethod
17033:     def should_use_simplified_methods(context: dict[str, Any]) -> bool:
17034:         """Check if simplified methods should be used."""
17035:         constraints = ResourceConstraints.get_constraints(context)
17036:         return constraints.get("use_simplified_methods", False)
17037:
17038:     @staticmethod
17039:     def should_skip_optional_analysis(context: dict[str, Any]) -> bool:
17040:         """Check if optional analysis should be skipped."""
17041:         constraints = ResourceConstraints.get_constraints(context)
17042:         return constraints.get("skip_optional_analysis", False)
17043:
17044:     @staticmethod
17045:     def get_entity_limit(context: dict[str, Any], default: int) -> int:
17046:         """Get entity limit with degradation applied."""
17047:         constraints = ResourceConstraints.get_constraints(context)
17048:         factor = constraints.get("entity_limit_factor", 1.0)
17049:         return int(default * factor)
17050:
17051:     @staticmethod
17052:     def get_embedding_dimensions(context: dict[str, Any], default: int) -> int:
17053:         """Get embedding dimensions with degradation applied."""
17054:         constraints = ResourceConstraints.get_constraints(context)
17055:         if constraints.get("reduce_embedding_dims", False):
17056:             return int(default * 0.5)
17057:         return default
17058:
17059:
```