

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 12
3: =====
4: Generated: 2025-12-07T06:17:19.681422
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: src/farfan_pipeline/core/calibration/data_structures-2.py
11: =====
12:
13: """
14: Calibration system data structures.
15:
16: These dataclasses define the EXACT structure of calibration outputs.
17: NO fields should be added or removed without updating this spec.
18:
19: Design Principles:
20: 1. Immutability: All dataclasses are frozen
21: 2. Validation: __post_init__ checks invariants
22: 3. Type Safety: Use type hints everywhere
23: 4. Serializability: Support to_dict() for JSON export
24: """
25: from dataclasses import dataclass, field
26: from typing import Any
27: from enum import Enum
28:
29:
30: class LayerID(str, Enum):
31:     """
32:     Exact identifier for each calibration layer.
33:
34:     These correspond to the 7 layers in the theoretical model.
35:     """
36:     BASE = "b"          # @b - Intrinsic quality (COMPLETE)
37:     UNIT = "u"          # @u - PDT quality
38:     QUESTION = "q"      # @q - Question compatibility
39:     DIMENSION = "d"    # @d - Dimension compatibility
40:     POLICY = "p"        # @p - Policy area compatibility
41:     CONGRUENCE = "C"   # @C - Ensemble validity
42:     CHAIN = "chain"    # @chain - Data flow integrity
43:     META = "m"          # @m - Governance
44:
45:
46: @dataclass(frozen=True)
47: class LayerScore:
48:     """
49:     Single layer evaluation result.
50:
51:     This represents the output of evaluating ONE layer for ONE subject.
52:
53:     Attributes:
54:         layer: Which layer this score belongs to
55:         score: Numerical score in [0.0, 1.0]
56:         components: Breakdown of sub-scores (e.g., for @u: {S, M, I, P})
```

```
57:     rationale: Human-readable explanation of the score
58:     metadata: Additional debug/audit information
59:
60:     Example:
61:         LayerScore(
62:             layer=LayerID.UNIT,
63:             score=0.75,
64:             components={"S": 0.8, "M": 0.7, "I": 0.75, "P": 0.75},
65:             rationale="Unit quality: robusto (S=0.80, M=0.70, I=0.75, P=0.75)",
66:             metadata={"aggregation_method": "geometric_mean"}
67:         )
68:     """
69:     layer: LayerID
70:     score: float # MUST be in [0.0, 1.0]
71:     components: dict[str, float] = field(default_factory=dict)
72:     rationale: str = ""
73:     metadata: dict[str, Any] = field(default_factory=dict)
74:
75:     def __post_init__(self):
76:         """Validate score is in valid range."""
77:         if not 0.0 <= self.score <= 1.0:
78:             raise ValueError(
79:                 f"Layer {self.layer.value} score {self.score} out of range [0.0, 1.0]"
80:             )
81:
82:     def to_dict(self) -> dict:
83:         """Export as dictionary for JSON serialization."""
84:         return {
85:             "layer": self.layer.value,
86:             "score": self.score,
87:             "components": self.components,
88:             "rationale": self.rationale,
89:             "metadata": self.metadata,
90:         }
91:
92:
93: @dataclass(frozen=True)
94: class ContextTuple:
95:     """
96:         Execution context for a micro-question: ctx = (Q, D, P, U).
97:
98:         This is the (Q, D, P, U) tuple that defines WHERE a method is being used.
99:         The context determines which compatibility scores apply.
100:
101:     Attributes:
102:         question_id: e.g., "Q001", "Q031" (from questionnaire monolith)
103:         dimension: e.g., "DIM01" (canonical code, not "D1")
104:         policy_area: e.g., "PA01" (canonical code, not "P1")
105:         unit_quality: Pre-computed U score from PDT analysis, range [0.0, 1.0]
106:
107:     Example:
108:         ContextTuple(
109:             question_id="Q001",
110:             dimension="DIM01",
111:             policy_area="PA01",
112:             unit_quality=0.75
```

```
113:         )
114:         """
115:         question_id: str
116:         dimension: str
117:         policy_area: str
118:         unit_quality: float
119:
120:     def __post_init__(self):
121:         """Validate canonical notation and ranges."""
122:         # Validate dimension uses canonical notation
123:         if not self.dimension.startswith("DIM"):
124:             raise ValueError(
125:                 f"Dimension must use canonical code (DIM01-DIM06), got {self.dimension}"
126             )
127:
128:         # Validate policy area uses canonical notation
129:         if not self.policy_area.startswith("PA"):
130:             raise ValueError(
131:                 f"Policy must use canonical code (PA01-PA10), got {self.policy_area}"
132             )
133:
134:         # Validate question ID format
135:         if not self.question_id.startswith("Q"):
136:             raise ValueError(
137:                 f"Question ID must start with 'Q', got {self.question_id}"
138             )
139:
140:         # Validate unit quality range
141:         if not 0.0 <= self.unit_quality <= 1.0:
142:             raise ValueError(
143:                 f"Unit quality must be in [0.0, 1.0], got {self.unit_quality}"
144             )
145:
146:     def to_dict(self) -> dict:
147:         """Export as dictionary."""
148:         return {
149:             "question_id": self.question_id,
150:             "dimension": self.dimension,
151:             "policy_area": self.policy_area,
152:             "unit_quality": self.unit_quality,
153:         }
154:
155:
156: @dataclass(frozen=True)
157: class CalibrationSubject:
158:     """
159:         Subject of calibration I = (M, v, \mathcal{G}, ctx).
160:
161:         This represents ONE method being evaluated in ONE context.
162:
163:         From theoretical model:
164:             - M: Method artifact (code)
165:             - v: Version
166:             - \mathcal{G}: Computational graph (how methods connect)
167:             - G: Interplay subgraph (methods working together)
168:             - ctx: Context tuple (Q, D, P, U)
```

```
169:  
170:     Attributes:  
171:         method_id: e.g., "pattern_extractor_v2"  
172:         method_version: e.g., "v2.1.0"  
173:         graph_config: Hash of the computational graph \\\n  
174:         subgraph_id: Identifier for the interplay subgraph G  
175:         context: The (Q, D, P, U) context  
176:  
177:     Example:  
178:         CalibrationSubject(  
179:             method_id="pattern_extractor_v2",  
180:             method_version="v2.1.0",  
181:             graph_config="abc123def456",  
182:             subgraph_id="Q001_analyzer_validator",  
183:             context=ContextTuple(...)  
184:         )  
185:         """  
186:         method_id: str  
187:         method_version: str  
188:         graph_config: str  
189:         subgraph_id: str  
190:         context: ContextTuple  
191:  
192:     def to_dict(self) -> dict:  
193:         """Export as dictionary."""  
194:         return {  
195:             "method_id": self.method_id,  
196:             "method_version": self.method_version,  
197:             "graph_config": self.graph_config,  
198:             "subgraph_id": self.subgraph_id,  
199:             "context": self.context.to_dict(),  
200:         }  
201:  
202:  
203: @dataclass(frozen=True)  
204: class CompatibilityMapping:  
205:     """  
206:         Defines how compatible a method is with questions/dimensions/policies.  
207:  
208:         This implements the Q_f, D_f, P_f functions from the theoretical model.  
209:  
210:         Compatibility Scores (from theoretical model):  
211:             1.0 = Primary (designed specifically for this context)  
212:             0.7 = Secondary (works well, but not optimal)  
213:             0.3 = Compatible (can work, limited effectiveness)  
214:             0.1 = Undeclared (penalty, not validated for this context)  
215:  
216:         Example:  
217:             CompatibilityMapping(  
218:                 method_id="pattern_extractor_v2",  
219:                 questions={"Q001": 1.0, "Q031": 0.7, "Q091": 0.3},  
220:                 dimensions={"DIM01": 1.0, "DIM03": 0.7},  
221:                 policies={"PA01": 1.0, "PA10": 0.7}  
222:             )  
223:             """  
224:             method_id: str
```

```
225:     questions: dict[str, float]    # question_id -> score {1.0, 0.7, 0.3, 0.1}
226:     dimensions: dict[str, float]  # dimension_code -> score
227:     policies: dict[str, float]    # policy_code -> score
228:
229:     def get_question_score(self, question_id: str) -> float:
230:         """
231:             Get compatibility score for a question.
232:
233:             Returns 0.1 (penalty) if question not declared.
234:         """
235:         return self.questions.get(question_id, 0.1)
236:
237:     def get_dimension_score(self, dimension: str) -> float:
238:         """
239:             Get compatibility score for a dimension.
240:
241:             Returns 0.1 (penalty) if dimension not declared.
242:         """
243:         return self.dimensions.get(dimension, 0.1)
244:
245:     def get_policy_score(self, policy: str) -> float:
246:         """
247:             Get compatibility score for a policy area.
248:
249:             Returns 0.1 (penalty) if policy not declared.
250:         """
251:         return self.policies.get(policy, 0.1)
252:
253:     def check_anti_universality(self, threshold: float = 0.9) -> bool:
254:         """
255:             Check Anti-Universality Theorem compliance.
256:
257:             The theorem states: NO method can have average compatibility <= 0.9
258:             across ALL questions, dimensions, AND policies simultaneously.
259:
260:             Returns:
261:                 True if compliant (method is NOT universal)
262:                 False if violation detected
263:         """
264:         if not self.questions or not self.dimensions or not self.policies:
265:             return True  # Incomplete mapping, cannot be universal
266:
267:         avg_q = sum(self.questions.values()) / len(self.questions)
268:         avg_d = sum(self.dimensions.values()) / len(self.dimensions)
269:         avg_p = sum(self.policies.values()) / len(self.policies)
270:
271:         is_universal = (avg_q >= threshold and
272:                         avg_d >= threshold and
273:                         avg_p >= threshold)
274:
275:         return not is_universal
276:
277:     def to_dict(self) -> dict:
278:         """Export as dictionary."""
279:         return {
280:             "method_id": self.method_id,
```

```
281:         "questions": self.questions,
282:         "dimensions": self.dimensions,
283:         "policies": self.policies,
284:     }
285:
286:
287: @dataclass(frozen=True)
288: class InteractionTerm:
289:     """
290:         Represents a synergy between two layers in Choquet aggregation.
291:
292:         Formula:  $a_{\text{layer}_1} \wedge a_{\text{layer}_2} \rightarrow \min(a_{\text{layer}_1}, a_{\text{layer}_2})$ 
293:
294:         This captures the "weakest link" principle: the contribution of the
295:         interaction is limited by whichever layer scored lower.
296:
297:         Standard Interactions (from theoretical model):
298:             (@u, @chain): weight=0.15, "Plan quality only matters with sound wiring"
299:             (@chain, @C): weight=0.12, "Ensemble validity requires chain integrity"
300:             (@q, @d): weight=0.08, "Question-dimension alignment synergy"
301:             (@d, @p): weight=0.05, "Dimension-policy coherence synergy"
302:
303:         Example:
304:             InteractionTerm(
305:                 layer_1=LayerID.UNIT,
306:                 layer_2=LayerID.CHAIN,
307:                 weight=0.15,
308:                 rationale="Plan quality only matters with sound wiring"
309:             )
310:             """
311:             layer_1: LayerID
312:             layer_2: LayerID
313:             weight: float # a_{\text{layer}} coefficient
314:             rationale: str # Why this interaction exists
315:
316:         def compute(self, scores: dict[LayerID, float]) -> float:
317:             """
318:                 Compute interaction contribution.
319:
320:                 Formula:  $a_{\text{layer}_1} \wedge a_{\text{layer}_2} \rightarrow \min(a_{\text{layer}_1}, a_{\text{layer}_2})$ 
321:
322:             Args:
323:                 scores: Dictionary mapping LayerID to score
324:
325:             Returns:
326:                 Interaction contribution (can be 0 if layer missing)
327:             """
328:             score_1 = scores.get(self.layer_1, 0.0)
329:             score_2 = scores.get(self.layer_2, 0.0)
330:             return self.weight * min(score_1, score_2)
331:
332:         def to_dict(self) -> dict:
333:             """Export as dictionary."""
334:             return {
335:                 "layer_1": self.layer_1.value,
336:                 "layer_2": self.layer_2.value,
```

```
337:         "weight": self.weight,
338:         "rationale": self.rationale,
339:     }
340:
341:
342: @dataclass(frozen=True)
343: class CalibrationResult:
344:     """
345:         Complete calibration output for a subject I.
346:
347:         This is the FINAL result of the calibration pipeline.
348:
349:         Formula: Cal(I) = if a_â\204\223•x_â\204\223 + if a_â\204\223k•min(x_â\204\223, x_k)
350:
351:         Attributes:
352:             subject: The calibration subject I = (M, v, i\223, G, ctx)
353:             layer_scores: Individual scores for each layer
354:             linear_contribution: if a_â\204\223 • x_â\204\223
355:             interaction_contribution: if a_â\204\223k • min(x_â\204\223, x_k)
356:             final_score: Cal(I) = linear + interaction â\210\210 [0.0, 1.0]
357:             computation_metadata: Timestamps, hashes, config_hash, etc.
358:
359:         Example:
360:             CalibrationResult(
361:                 subject=CalibrationSubject(...),
362:                 layer_scores={
363:                     LayerID.BASE: LayerScore(..., score=0.9),
364:                     LayerID.UNIT: LayerScore(..., score=0.75),
365:                     ...
366:                 },
367:                 linear_contribution=0.65,
368:                 interaction_contribution=0.15,
369:                 final_score=0.80,
370:                 computation_metadata={
371:                     "config_hash": "abc123",
372:                     "timestamp": "2025-11-11T10:30:00Z"
373:                 }
374:             )
375:             """
376:             subject: CalibrationSubject
377:             layer_scores: dict[LayerID, LayerScore]
378:             linear_contribution: float
379:             interaction_contribution: float
380:             final_score: float
381:             computation_metadata: dict[str, Any] = field(default_factory=dict)
382:
383:         def __post_init__(self):
384:             """Validate calibration result integrity."""
385:             # Validate final score range
386:             if not 0.0 <= self.final_score <= 1.0:
387:                 raise ValueError(
388:                     f"Final calibration score {self.final_score} out of range [0.0, 1.0]"
389:                 )
390:
391:             # Verify linear + interaction = final (within numerical tolerance)
392:             computed = self.linear_contribution + self.interaction_contribution
```

```
393:         if abs(computed - self.final_score) > 1e-6:
394:             raise ValueError(
395:                 f"Final score {self.final_score} != "
396:                 f"linear {self.linear_contribution} + "
397:                 f"interaction {self.interaction_contribution} = {computed}"
398:             )
399:
400:     # Verify all layer scores are in valid range
401:     for layer_id, layer_score in self.layer_scores.items():
402:         if not 0.0 <= layer_score.score <= 1.0:
403:             raise ValueError(
404:                 f"Layer {layer_id.value} score {layer_score.score} out of range"
405:             )
406:
407:     def to_certificate_dict(self) -> dict:
408:         """
409:             Export as a calibration certificate for auditing.
410:
411:             This is the format that gets saved to audit logs and can be
412:             used to reproduce the calibration result.
413:         """
414:         return {
415:             "certificate_version": "1.0",
416:             "method": {
417:                 "id": self.subject.method_id,
418:                 "version": self.subject.method_version,
419:                 "graph_config": self.subject.graph_config,
420:                 "subgraph_id": self.subject.subgraph_id,
421:             },
422:             "context": self.subject.context.to_dict(),
423:             "layer_scores": {
424:                 layer_id.value: layer_score.to_dict()
425:                 for layer_id, layer_score in self.layer_scores.items()
426:             },
427:             "aggregation": {
428:                 "linear_contribution": self.linear_contribution,
429:                 "interaction_contribution": self.interaction_contribution,
430:                 "final_score": self.final_score,
431:                 "formula": "Cal(I) = If a_â\204\223•x_â\204\223 + If a_â\204\223kâ\min(x_â\204\223, x_k)",
432:             },
433:             "metadata": self.computation_metadata,
434:         }
435:
436:     def to_dict(self) -> dict:
437:         """Export as dictionary."""
438:         return self.to_certificate_dict()
439:
440:
441:
442: =====
443: FILE: src/farfan_pipeline/core/calibration/data_structures.py
444: =====
445:
446: """
447: Three-Pillar Calibration System - Core Data Structures
448:
```

```
449: This module defines the fundamental data structures for the calibration system
450: as specified in the SUPERPROMPT Three-Pillar Calibration System.
451:
452: Spec compliance: Section 1 (Core Objects), Section 7 (Certificates)
453: """
454:
455: from dataclasses import dataclass, field
456: from typing import Dict, Any, Optional, List, Set, Tuple
457: from enum import Enum
458:
459:
460: class CalibrationConfigError(Exception):
461:     """
462:         Raised when calibration configuration violates mathematical constraints.
463:
464:         This error indicates:
465:             - Fusion weights don't sum to valid range
466:             - Weight constraints violated (must be ≥ 0)
467:             - Invalid layer configuration
468:             - Misconfigured calibration parameters
469:
470:         SIN_CARRETA Policy: Fail loudly on misconfiguration, never silently clamp.
471:     """
472:     pass
473:
474:
475: class LayerType(Enum):
476:     """Eight fixed calibration layers - NO RENAMING ALLOWED"""
477:     BASE = "@b"                      # Intrinsic quality
478:     CHAIN = "@chain"                 # Chain compatibility
479:     UNIT = "@u"                     # Unit-of-analysis sensitivity
480:     QUESTION = "@q"                  # Question compatibility
481:     DIMENSION = "@d"                 # Dimension compatibility
482:     POLICY = "@p"                   # Policy compatibility
483:     INTERPLAY = "@C"                # Interplay congruence
484:     META = "@m"                     # Meta/governance
485:
486:
487: class MethodRole(Enum):
488:     """Method roles with fixed required layer sets"""
489:     INGEST_PDM = "INGEST_PDM"
490:     STRUCTURE = "STRUCTURE"
491:     EXTRACT = "EXTRACT"
492:     SCORE_Q = "SCORE_Q"
493:     AGGREGATE = "AGGREGATE"
494:     REPORT = "REPORT"
495:     META_TOOL = "META_TOOL"
496:     TRANSFORM = "TRANSFORM"
497:
498:
499: # Role-based required layers (L_* from spec Section 4)
500: REQUIRED_LAYERS: Dict[MethodRole, Set[LayerType]] = {
501:     MethodRole.INGEST_PDM: {LayerType.BASE, LayerType.CHAIN, LayerType.UNIT, LayerType.META},
502:     MethodRole.STRUCTURE: {LayerType.BASE, LayerType.CHAIN, LayerType.UNIT, LayerType.META},
503:     MethodRole.EXTRACT: {LayerType.BASE, LayerType.CHAIN, LayerType.UNIT, LayerType.META},
504:     MethodRole.SCORE_Q: {LayerType.BASE, LayerType.CHAIN, LayerType.QUESTION, LayerType.DIMENSION},
```

```

505:     LayerType.POLICY, LayerType.INTERPLAY, LayerType.UNIT, LayerType.META},
506:     MethodRole.AGGREGATE: {LayerType.BASE, LayerType.CHAIN, LayerType.DIMENSION, LayerType.POLICY,
507:         LayerType.INTERPLAY, LayerType.META},
508:     MethodRole.REPORT: {LayerType.BASE, LayerType.CHAIN, LayerType.INTERPLAY, LayerType.META},
509:     MethodRole.META_TOOL: {LayerType.BASE, LayerType.CHAIN, LayerType.META},
510:     MethodRole.TRANSFORM: {LayerType.BASE, LayerType.CHAIN, LayerType.META},
511: }
512:
513:
514: @dataclass(frozen=True)
515: class Context:
516:     """
517:     Execution context: ctx = (Q, D, P, U)
518:
519:     Spec compliance: Definition 1.2
520:
521:     Q: Question ID or None
522:     D: Dimension ID (DIM01-DIM06)
523:     P: Policy area ID (PA01-PA10)
524:     U: Unit-of-analysis quality [0,1]
525:     """
526:     question_id: Optional[str] = None # Q à\210\210 Questions à\210ª {à\212¥}
527:     dimension_id: str = "DIM01" # D à\210\210 Dimensions
528:     policy_id: str = "PA01" # P à\210\210 Policies
529:     unit_quality: float = 0.85 # U à\210\210 [0,1]
530:
531:     def __post_init__(self):
532:         """Validate context constraints"""
533:         if self.unit_quality < 0.0 or self.unit_quality > 1.0:
534:             raise ValueError(f"unit_quality must be in [0,1], got {self.unit_quality}")
535:
536:         if self.dimension_id and not self.dimension_id.startswith("DIM"):
537:             raise ValueError(f"dimension_id must match DIM* pattern, got {self.dimension_id}")
538:
539:         if self.policy_id and not self.policy_id.startswith("PA"):
540:             raise ValueError(f"policy_id must match PA* pattern, got {self.policy_id}")
541:
542:
543: @dataclass
544: class ComputationGraph:
545:     """
546:     Computation graph: Î\223 = (V, E, T, S)
547:
548:     Spec compliance: Definition 1.1
549:
550:     V: finite set of method instance nodes
551:     E: directed edges (must be DAG)
552:     T: edge typing function
553:     S: node signature function
554:     """
555:     nodes: Set[str] = field(default_factory=set) # V
556:     edges: List[Tuple[str, str]] = field(default_factory=list) # E à\212\206 V à\227 V
557:     edge_types: Dict[Tuple[str, str], Dict[str, Any]] = field(default_factory=dict) # T
558:     node_signatures: Dict[str, Dict[str, Any]] = field(default_factory=dict) # S
559:
560:     def validate_dag(self) -> bool:

```

```
561:     """Axiom 1.1: Graph must be acyclic"""
562:     # Simple cycle detection via DFS
563:     visited = set()
564:     rec_stack = set()
565:
566:     def has_cycle(node: str) -> bool:
567:         visited.add(node)
568:         rec_stack.add(node)
569:
570:         for edge in self.edges:
571:             if edge[0] == node:
572:                 neighbor = edge[1]
573:                 if neighbor not in visited:
574:                     if has_cycle(neighbor):
575:                         return True
576:                 elif neighbor in rec_stack:
577:                     return True
578:
579:             rec_stack.remove(node)
580:         return False
581:
582:     for node in self.nodes:
583:         if node not in visited:
584:             if has_cycle(node):
585:                 return False
586:     return True
587:
588:
589: @dataclass
590: class InterplaySubgraph:
591:     """
592:         Valid interplay: G = (V_G, E_G) \u21d2\206 \u223
593:
594:         Spec compliance: Definition 2.1
595:
596:         Must satisfy:
597:             1. Single target property
598:             2. Declared fusion rule
599:             3. Type compatibility
600:     """
601:     nodes: Set[str]
602:     edges: List[Tuple[str, str]]
603:     target_output: str
604:     fusion_rule: str
605:     compatible: bool = True
606:
607:
608: @dataclass(frozen=True)
609: class CalibrationCertificate:
610:     """
611:         Complete calibration certificate with audit trail.
612:
613:         Spec compliance: Section 7 (Definition 7.1)
614:
615:         MUST allow exact reconstruction of Cal(I) from contents.
616:         Property 7.1: No hidden behavior - all computations must appear here.
```

```
617: """
618: # Identity
619: instance_id: str
620: method_id: str
621: node_id: str
622: context: Context
623:
624: # Scores
625: intrinsic_score: float # x_@b
626: layer_scores: Dict[str, float] # All x_â\204\223(I)
627: calibrated_score: float # Cal(I)
628:
629: # Transparency
630: fusion_formula: Dict[str, Any] # symbolic, expanded, computation_trace
631: parameter_provenance: Dict[str, Dict[str, Any]] # Where each parameter came from
632: evidence_trail: Dict[str, Any] # Evidence used for layer computations
633:
634: # Integrity
635: config_hash: str # SHA256 of all config files
636: graph_hash: str # SHA256 of computation graph
637:
638: # Validation
639: validation_checks: Dict[str, Any] = field(default_factory=dict)
640: sensitivity_analysis: Dict[str, Any] = field(default_factory=dict)
641:
642: # Audit
643: timestamp: str = ""
644: validator_version: str = "1.0.0"
645:
646: def __post_init__(self):
647:     """Validate certificate constraints"""
648:     # Boundedness check
649:     if not (0.0 <= self.calibrated_score <= 1.0):
650:         raise ValueError(f"calibrated_score must be in [0,1], got {self.calibrated_score}")
651:
652:     if not (0.0 <= self.intrinsic_score <= 1.0):
653:         raise ValueError(f"intrinsic_score must be in [0,1], got {self.intrinsic_score}")
654:
655:     for layer, score in self.layer_scores.items():
656:         if not (0.0 <= score <= 1.0):
657:             raise ValueError(f"layer_scores[{layer}] must be in [0,1], got {score}")
658:
659:
660: @dataclass
661: class CalibrationSubject:
662: """
663: Calibration subject: I = (M, v, Î\223, G, ctx)
664:
665: Spec compliance: Definition 1.3
666:
667: M: method artifact
668: v: node instance
669: Î\223: containing graph
670: G: interplay subgraph (or None)
671: ctx: execution context
672: """
```

```
673:     method_id: str    # M (canonical method ID)
674:     node_id: str      # v à\210\210 V
675:     graph: ComputationGraph  # î\223
676:     interplay: Optional[InterplaySubgraph]  # G
677:     context: Context   # ctx
678:
679:     # Additional metadata
680:     role: Optional[MethodRole] = None
681:     active_layers: Set[LayerType] = field(default_factory=set)
682:
683:
684: @dataclass
685: class EvidenceStore:
686: """
687:     Storage for evidence used in calibration computations.
688:     All evidence must be traceable and auditable.
689: """
690:     pdt_structure: Dict[str, Any] = field(default_factory=dict)
691:     pdm_metrics: Dict[str, Any] = field(default_factory=dict)
692:     runtime_metrics: Dict[str, Any] = field(default_factory=dict)
693:     test_results: Dict[str, Any] = field(default_factory=dict)
694:     deployment_history: Dict[str, Any] = field(default_factory=dict)
695:
696:     def get_evidence(self, key: str, default: Any = None) -> Any:
697:         """Retrieve evidence by key"""
698:         for store in [self.pdt_structure, self.pdm_metrics, self.runtime_metrics,
699:                      self.test_results, self.deployment_history]:
700:             if key in store:
701:                 return store[key]
702:         return default
703:
704:
705:
706: =====
707: FILE: src/farfan_pipeline/core/calibration/decorators.py
708: =====
709:
710: """Calibration decorators using centralized ParameterLoaderV2."""
711:
712: import functools
713: import logging
714: from collections.abc import Callable
715: from typing import Any
716:
717: from farfan_pipeline.core.parameters import ParameterLoaderV2
718:
719: logger = logging.getLogger(__name__)
720:
721:
722: def calibrated_method(method_id: str) -> Callable:
723: """
724:     Decorator to apply calibration to a method using centralized ParameterLoaderV2.
725:
726:     Future: Will invoke CalibrationOrchestrator.calibrate(method_id, context) when available.
727:
728:     Args:
```

```
729:         method_id: Fully qualified method identifier for parameter lookup
730:
731:     Returns:
732:         Decorated function with calibration applied
733:     """
734:
735:     def decorator(func: Callable) -> Callable:
736:         @functools.wraps(func)
737:         def wrapper(*args: Any, **kwargs: Any) -> Any:
738:             calibration_params = ParameterLoaderV2.get_all(method_id)
739:
740:             logger.debug(
741:                 f"Calling calibrated method '{method_id}' with {len(calibration_params)} parameters"
742:             )
743:
744:             # Future: CalibrationOrchestrator.calibrate(method_id, context={
745: #                 "args": args,
746: #                 "kwargs": kwargs,
747: #                 "params": calibration_params
748: #             })
749:
750:             return func(*args, **kwargs)
751:
752:         return wrapper
753:
754:     return decorator
755:
756:
757:
758: =====
759: FILE: src/farfan_pipeline/core/calibration/engine.py
760: =====
761:
762: """
763: Three-Pillar Calibration System - Main Calibration Engine
764:
765: This module implements the main calibrate() function and fusion operator
766: as specified in the SUPERPROMPT Three-Pillar Calibration System.
767:
768: Spec compliance: Section 5 (Fusion Operator), Section 6 (Runtime Engine)
769: SIN_CARRETA Compliance: Pure fusion operator, fail-loudly on misconfiguration
770: """
771:
772: import json
773: import hashlib
774: from datetime import datetime, timezone
775: from pathlib import Path
776: from typing import Dict, Any, Optional
777: from farfan_pipeline.core.calibration.data_structures import (
778:     CalibrationCertificate, CalibrationSubject, Context,
779:     ComputationGraph, EvidenceStore, LayerType, MethodRole, REQUIRED_LAYERS,
780:     CalibrationConfigError
781: )
782: from farfan_pipeline.core.calibration.layer_computers import (
783:     compute_base_layer, compute_chain_layer, compute_unit_layer,
784:     compute_question_layer, compute_dimension_layer, compute_policy_layer,
```

```
785:     compute_interplay_layer, compute_meta_layer
786: )
787:
788:
789: class CalibrationEngine:
790:     """
791:         Main calibration engine implementing the three-pillar system.
792:
793:         Spec compliance: Section 7 (Runtime Engine & Certificate)
794:     """
795:
796:     def __init__(self, config_dir: str = None, monolith_path: str = None, catalog_path: str = None):
797:         """
798:             Initialize calibration engine and load configs.
799:
800:             SIN-CARRETA: Validates fusion weights at load time to fail fast.
801:             Three-Pillar System: Loads from intrinsic, contextual, and fusion configs.
802:
803:             Args:
804:                 config_dir: Path to config directory (defaults to ../config)
805:                 monolith_path: Path to questionnaire_monolith.json (defaults to ../data/questionnaire_monolith.json)
806:                 catalog_path: Path to canonical_method_catalog.json (defaults to ../config/canonical_method_catalog.json)
807:
808:             Raises:
809:                 CalibrationConfigError: If fusion weights violate constraints
810:             """
811:             if config_dir is None:
812:                 config_dir = Path(__file__).parent.parent / "config"
813:             else:
814:                 config_dir = Path(config_dir)
815:
816:             self.config_dir = config_dir
817:             self.intrinsic_config = self._load_json(config_dir / "intrinsic_calibration.json")
818:             self.contextual_config = self._load_json(config_dir / "contextual_parametrization.json")
819:             self.fusion_config = self._load_json(config_dir / "fusion_specification.json")
820:
821:             # Load canonical method catalog for role determination
822:             if catalog_path is None:
823:                 catalog_path = config_dir / "canonical_method_catalog.json"
824:             else:
825:                 catalog_path = Path(catalog_path)
826:             self.catalog = self._load_json(catalog_path)
827:             self._build_method_index()
828:
829:             # SIN-CARRETA: Validate fusion weights at load time
830:             self._validate_fusion_weights()
831:
832:             # Load questionnaire monolith using canonical loader
833:             # This ensures hash verification and immutability
834:             from saaaaa.core.orchestrator.factory import load_questionnaire
835:
836:             if monolith_path is None:
837:                 # Use default path from factory
838:                 canonical_q = load_questionnaire()
839:             else:
840:                 # Use specified path
```

```

841:         canonical_q = load_questionnaire(Path(monolith_path))
842:
843:         # Convert to dict for backward compatibility with calibration system
844:         # (CalibrationEngine expects dict, not CanonicalQuestionnaire)
845:         self.monolith = dict(canonical_q.data)
846:         self._questionnaire_hash = canonical_q.sha256 # Store for verification
847:
848:         # Compute config hash
849:         self.config_hash = self._compute_config_hash()
850:
851:     @staticmethod
852:     def _load_json(path: Path) -> Dict[str, Any]:
853:         """Load JSON file"""
854:         with open(path, 'r') as f:
855:             return json.load(f)
856:
857:     def _build_method_index(self) -> None:
858:         """
859:             Build index of methods from canonical catalog for fast role lookup.
860:
861:             Three-Pillar System: Uses canonical_method_catalog.json as single source.
862:         """
863:         self.method_index = {}
864:
865:         for layer_name, methods in self.catalog.get("layers", {}).items():
866:             for method_info in methods:
867:                 canonical_name = method_info.get("canonical_name", "")
868:                 method_name = method_info.get("method_name", "")
869:                 class_name = method_info.get("class_name", "")
870:                 layer = method_info.get("layer", "unknown")
871:
872:                 # Store method info with multiple lookup keys
873:                 for key in [canonical_name, method_name, f"{class_name}.{method_name}"]:
874:                     if key:
875:                         self.method_index[key] = {
876:                             "canonical_name": canonical_name,
877:                             "method_name": method_name,
878:                             "class_name": class_name,
879:                             "layer": layer,
880:                             "metadata": method_info
881:                         }
882:
883:     def _validate_fusion_weights(self) -> None:
884:         """
885:             Validate fusion weight constraints at config load time.
886:
887:             Per canonic_calibration_methods.md specification.
888:
889:             Constraints:
890:             1. All weights must be non-negative: a_â\204\223 â\211¥ 0, a_â\204\223k â\211¥ 0
891:             2. Total weight sum MUST equal 1: Îf(a_â\204\223) + Îf(a_â\204\223k) = 1 (tolerance 1e-9)
892:
893:             Raises:
894:                 CalibrationConfigError: If any weight constraint is violated
895:             """
896:             role_params_dict = self.fusion_config.get("role_fusion_parameters", {})

```

```

897:     TOLERANCE = 1e-9
898:
899:     for role_name, role_params in role_params_dict.items():
900:         linear_weights = role_params.get("linear_weights", {})
901:         interaction_weights = role_params.get("interaction_weights", {})
902:
903:         # Constraint 1: Non-negativity
904:         for layer, weight in linear_weights.items():
905:             if weight < 0:
906:                 raise CalibrationConfigError(
907:                     f"Negative weight for role={role_name}, layer={layer}: "
908:                     f"weight={weight}. All weights must be ≥ 0."
909:                 )
910:
911:         for pair, weight in interaction_weights.items():
912:             if weight < 0:
913:                 raise CalibrationConfigError(
914:                     f"Negative interaction weight for role={role_name}, pair={pair}: "
915:                     f"weight={weight}. All weights must be ≥ 0."
916:                 )
917:
918:         # Constraint 2: Must sum to exactly 1.0
919:         total_weight = sum(linear_weights.values()) + sum(interaction_weights.values())
920:         if abs(total_weight - 1.0) > TOLERANCE:
921:             raise CalibrationConfigError(
922:                 f"Weight sum must equal 1.0 for role={role_name}: "
923:                 f"total_weight={total_weight:.15f} (deviation: {abs(total_weight - 1.0):.15f}). "
924:                 f"Constraint: |a_{role_name}| + |a_{role_name}_pair| = 1.0 (tolerance {TOLERANCE})."
925:             )
926:
927:     def _compute_config_hash(self) -> str:
928:         """
929:             Compute SHA256 hash of all config files.
930:
931:             Spec compliance: Section 7 (audit_trail.config_hash)
932:         """
933:         hasher = hashlib.sha256()
934:
935:         # Hash all three pillar configs in sorted order
936:         for config in sorted([
937:             json.dumps(self.intrinsic_config, sort_keys=True),
938:             json.dumps(self.contextual_config, sort_keys=True),
939:             json.dumps(self.fusion_config, sort_keys=True),
940:         ]):
941:             hasher.update(config.encode('utf-8'))
942:
943:         return f"sha256:{hasher.hexdigest()}"
944:
945:     @staticmethod
946:     def _compute_graph_hash(graph: ComputationGraph) -> str:
947:         """
948:             Compute SHA256 hash of computation graph.
949:
950:             Spec compliance: Section 7 (audit_trail.graph_hash)
951:         """
952:         hasher = hashlib.sha256()

```

```
953:
954:     # Hash nodes and edges
955:     graph_repr = json.dumps({
956:         "nodes": sorted(list(graph.nodes)),
957:         "edges": sorted([list(e) for e in graph.edges])
958:     }, sort_keys=True)
959:
960:     hasher.update(graph_repr.encode('utf-8'))
961:     return f"sha256:{hasher.hexdigest()}"
962:
963:     def _determine_role(self, method_id: str) -> MethodRole:
964:         """
965:             Determine method role from method ID using canonical catalog metadata.
966:
967:             Three-Pillar System: Uses canonical_method_catalog.json for role inference.
968:             Mapping from catalog layer + method patterns to MethodRole enum.
969:
970:             Args:
971:                 method_id: Method identifier (canonical_name, method_name, or Class.method format)
972:
973:             Returns:
974:                 MethodRole enum value
975:
976:             Raises:
977:                 CalibrationConfigError: If method not found in catalog or role cannot be determined
978:             """
979:             # Look up method in catalog index
980:             method_info = self.method_index.get(method_id)
981:
982:             if not method_info:
983:                 # Try fallback patterns
984:                 for key, info in self.method_index.items():
985:                     if method_id in key or key in method_id:
986:                         method_info = info
987:                         break
988:
989:             if not method_info:
990:                 # Cannot calibrate unknown methods - fail loudly
991:                 raise CalibrationConfigError(
992:                     f"Method '{method_id}' not found in canonical_method_catalog.json. "
993:                     f"Cannot determine role for calibration. "
994:                     f"All calibrated methods must be registered in catalog.\n"
995:                     f"To resolve:\n"
996:                     f"  1. Add method to config/canonical_method_catalog.json with proper metadata\n"
997:                     f"  2. Run scripts/rigorous_calibration_triage.py to generate intrinsic calibration\n"
998:                     f"  3. Ensure method has correct layer, role, and signature information"
999:                 )
1000:
1001:             # Determine role from layer + method name patterns (per canonic_calibration_methods.md)
1002:             layer = method_info.get("layer", "unknown")
1003:             method_name = method_info.get("method_name", "").lower()
1004:
1005:             # Role mapping based on layer and method semantics
1006:             # Per L_* specification in canonic_calibration_methods.md
1007:             if layer == "ingestion" or "ingest" in method_name or "pdm" in method_name:
1008:                 return MethodRole.INGEST_PDM
```

```
1009:         elif "structure" in method_name or "parse" in method_name:
1010:             return MethodRole.STRUCTURE
1011:         elif "extract" in method_name:
1012:             return MethodRole.EXTRACT
1013:         elif "score" in method_name or "question" in method_name or layer == "analyzer":
1014:             return MethodRole.SCORE_Q
1015:         elif "aggregate" in method_name or "combine" in method_name:
1016:             return MethodRole.AGGREGATE
1017:         elif "report" in method_name or "format" in method_name:
1018:             return MethodRole.REPORT
1019:         elif "transform" in method_name or "normalize" in method_name or "convert" in method_name:
1020:             return MethodRole.TRANSFORM
1021:         else:
1022:             # Default to META_TOOL for utility/orchestrator methods
1023:             return MethodRole.META_TOOL
1024:
1025:     def _detect_interplay(self, graph: ComputationGraph, node_id: str) -> Optional[Any]:
1026:         """
1027:             Detect interplay patterns from computation graph.
1028:
1029:             Three-Pillar System: Interplays are DECLARED in config, not auto-detected.
1030:
1031:             Per canonic_calibration_methods.md Section 1.3:
1032:             - "An interplay G is valid only if all nodes share a single declared target output"
1033:             - "A fusion rule is declared in config"
1034:             - "Do not infer ensembles implicitly"
1035:
1036:             This method checks if the node participates in any declared interplay
1037:             from the contextual config.
1038:
1039:             Args:
1040:                 graph: Computation graph
1041:                 node_id: Node identifier
1042:
1043:             Returns:
1044:                 Interplay subgraph if node participates in one, None otherwise
1045:             """
1046:             # Per specification: interplays are declared in contextual config, not inferred
1047:             # Check contextual_parametrization.json for declared interplays
1048:             interplay_defs = self.contextual_config.get("interplay_definitions", {})
1049:
1050:             for interplay_id, interplay_spec in interplay_defs.items():
1051:                 # Check if node_id is in this interplay's participant list
1052:                 participants = interplay_spec.get("participants", [])
1053:                 if node_id in participants:
1054:                     # Node participates in this declared interplay
1055:                     # Return interplay specification
1056:                     return {
1057:                         "interplay_id": interplay_id,
1058:                         "participants": participants,
1059:                         "target_output": interplay_spec.get("target_output"),
1060:                         "fusion_rule": interplay_spec.get("fusion_rule"),
1061:                         "declared": True
1062:                     }
1063:
1064:             # Node does not participate in any declared interplay
```

```
1065:         # This is normal - most nodes don't participate in interplays
1066:         return None
1067:
1068:     def _compute_layer_scores(
1069:         self,
1070:         subject: CalibrationSubject,
1071:         evidence: EvidenceStore
1072:     ) -> Dict[str, float]:
1073:         """
1074:             Compute all layer scores for calibration subject.
1075:
1076:             Spec compliance: Section 3 (all layers)
1077:             """
1078:         ctx = subject.context
1079:         scores = {}
1080:
1081:         # @b: Base layer (always required)
1082:         scores[LayerType.BASE.value] = compute_base_layer(
1083:             subject.method_id, self.intrinsic_config
1084:         )
1085:
1086:         # @chain: Chain compatibility (always required for non-META roles)
1087:         scores[LayerType.CHAIN.value] = compute_chain_layer(
1088:             subject.node_id, subject.graph, self.contextual_config
1089:         )
1090:
1091:         # @u: Unit-of-analysis
1092:         if subject.role:
1093:             scores[LayerType.UNIT.value] = compute_unit_layer(
1094:                 subject.method_id, subject.role, ctx.unit_quality, self.contextual_config
1095:             )
1096:
1097:         # @q: Question compatibility
1098:         scores[LayerType.QUESTION.value] = compute_question_layer(
1099:             subject.method_id, ctx.question_id, self.monolith, self.contextual_config
1100:         )
1101:
1102:         # @d: Dimension compatibility
1103:         scores[LayerType.DIMENSION.value] = compute_dimension_layer(
1104:             subject.method_id, ctx.dimension_id, self.contextual_config
1105:         )
1106:
1107:         # @p: Policy compatibility
1108:         scores[LayerType.POLICY.value] = compute_policy_layer(
1109:             subject.method_id, ctx.policy_id, self.contextual_config
1110:         )
1111:
1112:         # @C: Interplay congruence
1113:         scores[LayerType.INTERPLAY.value] = compute_interplay_layer(
1114:             subject.interplay, self.contextual_config
1115:         )
1116:
1117:         # @m: Meta/governance
1118:         meta_evidence = {
1119:             "formula_export_valid": True,
1120:             "trace_complete": True,
```

```

1121:         "logs_conform_schema": True,
1122:         "version_tagged": True,
1123:         "config_hash_matches": True,
1124:         "signature_valid": True,
1125:         "runtime_ms": evidence.runtime_metrics.get("runtime_ms", 100)
1126:     }
1127:     scores[LayerType.META.value] = compute_meta_layer(
1128:         meta_evidence, self.contextual_config
1129:     )
1130:
1131:     return scores
1132:
1133:     def _apply_fusion(
1134:         self,
1135:         role: MethodRole,
1136:         layer_scores: Dict[str, float]
1137:     ) -> tuple[float, Dict[str, Any]]:
1138:         """
1139:             Apply pure fusion operator to combine layer scores.
1140:
1141:             Spec compliance: Section 5 (Fusion Operator)
1142:             SIN_CARRETA Compliance: Pure mathematical formula, no clamping/normalization
1143:
1144:             Formula:  $\text{Cal}(I) = \sum_{k=1}^K a_k x_k + \sum_{k=1}^K a_k \min(x_k, 1)$ 
1145:
1146:             Weight constraints (enforced at load time):
1147:             - All weights  $a_k \geq 0$ ,  $a_k \leq 1$ 
1148:             -  $\sum_{k=1}^K a_k = 1$  (ensures boundedness)
1149:
1150:             Returns:
1151:                 (calibrated_score, fusion_details)
1152:
1153:             Raises:
1154:                 CalibrationConfigError: If score violates [0,1] bounds (weight misconfiguration)
1155:         """
1156:         role_params = self.fusion_config["role_fusion_parameters"].get(
1157:             role.value,
1158:             self.fusion_config["default_fallback"]
1159:         )
1160:
1161:         linear_weights = role_params["linear_weights"]
1162:         interaction_weights = role_params.get("interaction_weights", {})
1163:
1164:         # Compute linear terms
1165:         linear_sum = 0.0
1166:         linear_trace = []
1167:
1168:         for layer_key, weight in linear_weights.items():
1169:             if layer_key in layer_scores:
1170:                 contribution = weight * layer_scores[layer_key]
1171:                 linear_sum += contribution
1172:                 linear_trace.append({
1173:                     "layer": layer_key,
1174:                     "weight": weight,
1175:                     "score": layer_scores[layer_key],
1176:                     "contribution": contribution

```

```

1177:         })
1178:
1179:     # Compute interaction terms
1180:     interaction_sum = 0.0
1181:     interaction_trace = []
1182:
1183:     for pair_key, weight in interaction_weights.items():
1184:         # Parse "(layer1, layer2)" format
1185:         pair_str = pair_key.strip("()")
1186:         layer1, layer2 = [l.strip() for l in pair_str.split(",")]
1187:
1188:         if layer1 in layer_scores and layer2 in layer_scores:
1189:             min_score = min(layer_scores[layer1], layer_scores[layer2])
1190:             contribution = weight * min_score
1191:             interaction_sum += contribution
1192:             interaction_trace.append({
1193:                 "pair": pair_key,
1194:                 "weight": weight,
1195:                 "layer1_score": layer_scores[layer1],
1196:                 "layer2_score": layer_scores[layer2],
1197:                 "min_score": min_score,
1198:                 "contribution": contribution
1199:             })
1200:
1201:     # Total calibrated score (PURE FUSION - no clamping or normalization)
1202:     calibrated_score = linear_sum + interaction_sum
1203:
1204:     # SIN_CARRETA: Fail loudly on weight misconfiguration
1205:     # NEVER clamp or normalize - that would hide misconfiguration
1206:     if calibrated_score < 0.0 or calibrated_score > 1.0:
1207:         total_weight = sum(linear_weights.values()) + sum(interaction_weights.values())
1208:         raise CalibrationConfigError(
1209:             f"Fusion weights misconfigured for role {role.value}: "
1210:             f"total_weight={total_weight:.6f} produced calibrated_score={calibrated_score:.6f}. "
1211:             f"Score must be in [0,1]. Weight constraints violated. "
1212:             f"Check fusion_specification.json and ensure if(a_&\204\223) + if(a_&\204\223k) \211 1."
1213:         )
1214:
1215:     fusion_details = {
1216:         "symbolic": "if(a_&\204\223&x_&\204\223) + if(a_&\204\223k&min(x_&\204\223,x_k))",
1217:         "linear_terms": linear_trace,
1218:         "interaction_terms": interaction_trace,
1219:         "linear_sum": linear_sum,
1220:         "interaction_sum": interaction_sum,
1221:         "total": calibrated_score
1222:     }
1223:
1224:     return calibrated_score, fusion_details
1225:
1226: def calibrate(
1227:     self,
1228:     method_id: str,
1229:     node_id: str,
1230:     graph: ComputationGraph,
1231:     context: Context,
1232:     evidence_store: EvidenceStore

```

```
1233:     ) -> CalibrationCertificate:
1234:         """
1235:             Main calibration function.
1236:
1237:             Spec compliance: Section 7 (Runtime Engine)
1238:
1239:             Args:
1240:                 method_id: Canonical method ID
1241:                 node_id: Node identifier in graph
1242:                 graph: Computation graph
1243:                 context: Execution context
1244:                 evidence_store: Evidence for calibration
1245:
1246:             Returns:
1247:                 CalibrationCertificate with complete audit trail
1248:
1249:             Raises:
1250:                 ValueError: If validation fails
1251:             """
1252:             # Validate graph is DAG
1253:             if not graph.validate_dag():
1254:                 raise ValueError("Graph contains cycles - must be DAG")
1255:
1256:             # Determine role
1257:             role = self._determine_role(method_id)
1258:
1259:             # SIN_CARRETA: Detect interplay from graph (fail if not implemented)
1260:             interplay = self._detect_interplay(graph, node_id)
1261:
1262:             # Create calibration subject
1263:             subject = CalibrationSubject(
1264:                 method_id=method_id,
1265:                 node_id=node_id,
1266:                 graph=graph,
1267:                 interplay=interplay,
1268:                 context=context,
1269:                 role=role
1270:             )
1271:
1272:             # Validate layer completeness
1273:             required = REQUIRED_LAYERS.get(role, set())
1274:
1275:             # Compute layer scores
1276:             layer_scores = self._compute_layer_scores(subject, evidence_store)
1277:
1278:             # Check all required layers are present
1279:             missing_layers = [layer for layer in required if layer.value not in layer_scores]
1280:             if missing_layers:
1281:                 raise ValueError(
1282:                     f"Missing required layers for role {role.value}: "
1283:                     f"[{layer.value for layer in missing_layers}]"
1284:                 )
1285:
1286:             # Apply fusion
1287:             calibrated_score, fusion_details = self._apply_fusion(role, layer_scores)
1288:
```

```
1289:     # Build parameter provenance
1290:     role_params = self.fusion_config["role_fusion_parameters"].get(
1291:         role.value,
1292:         self.fusion_config["default_fallback"]
1293:     )
1294:
1295:     parameter_provenance = {
1296:         "fusion_weights": {
1297:             "source": "fusion_specification.json",
1298:             "role": role.value,
1299:             "linear_weights": role_params["linear_weights"],
1300:             "interaction_weights": role_params.get("interaction_weights", {})
1301:         },
1302:         "intrinsic_calibration": {
1303:             "source": "intrinsic_calibration.json",
1304:             "method_id": method_id
1305:         }
1306:     }
1307:
1308:     # Build evidence trail
1309:     evidence_trail = {
1310:         "pdt_metrics": evidence_store.pdt_structure,
1311:         "runtime_metrics": evidence_store.runtime_metrics,
1312:         "layer_computations": layer_scores
1313:     }
1314:
1315:     # Create certificate
1316:     certificate = CalibrationCertificate(
1317:         instance_id=f"{method_id}@{node_id}",
1318:         method_id=method_id,
1319:         node_id=node_id,
1320:         context=context,
1321:         intrinsic_score=layer_scores.get(LayerType.BASE.value, 0.0),
1322:         layer_scores=layer_scores,
1323:         calibrated_score=calibrated_score,
1324:         fusion_formula=fusion_details,
1325:         parameter_provenance=parameter_provenance,
1326:         evidence_trail=evidence_trail,
1327:         config_hash=self.config_hash,
1328:         graph_hash=self._compute_graph_hash(graph),
1329:         timestamp=datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z'),
1330:         validator_version="1.0.0"
1331:     )
1332:
1333:     return certificate
1334:
1335:
1336: # Convenience function
1337: def calibrate(
1338:     method_id: str,
1339:     node_id: str,
1340:     graph: ComputationGraph,
1341:     context: Context,
1342:     evidence_store: EvidenceStore,
1343:     config_dir: Optional[str] = None,
1344:     monolith_path: Optional[str] = None
```

```
1345: ) -> CalibrationCertificate:  
1346:     """  
1347:     Calibrate a method instance.  
1348:  
1349:     Spec compliance: Section 7  
1350:  
1351:     This is the single authoritative calibration entry point.  
1352:     """  
1353:     engine = CalibrationEngine(config_dir=config_dir, monolith_path=monolith_path)  
1354:     return engine.calibrate(method_id, node_id, graph, context, evidence_store)  
1355:  
1356:  
1357:  
1358: =====  
1359: FILE: src/farfan_pipeline/core/calibration/intrinsic_calibration_loader.py  
1360: =====  
1361:  
1362: """  
1363: intrinsic_calibration_loader.py - Single source loader for @b-layer intrinsic calibration  
1364:  
1365: This module provides the ONLY interface for loading intrinsic calibration data.  
1366: Enforces strict @b-only access with fallback behavior:  
1367: - pending \206\222 @b = 0.5 (neutral baseline)  
1368: - excluded \206\222 @b = None (causes method skip)  
1369: - none \206\222 @b = 0.3 (low confidence with warning)  
1370: - computed \206\222 actual @b values from JSON  
1371:  
1372: CRITICAL: This is the single source of truth for intrinsic calibration.  
1373: """  
1374: import json  
1375: import logging  
1376: from dataclasses import dataclass  
1377: from pathlib import Path  
1378:  
1379: logger = logging.getLogger(__name__)  
1380:  
1381:  
1382: @dataclass  
1383: class IntrinsicCalibration:  
1384:     """Intrinsic calibration data for a single method."""  
1385:     method_id: str  
1386:     intrinsic_score: tuple[float, float]  
1387:     b_theory: float  
1388:     b_impl: float  
1389:     b_deploy: float  
1390:     calibration_status: str  
1391:     layer: str  
1392:     last_updated: str  
1393:  
1394:     def get_composite_b(self) -> float:  
1395:         """Calculate composite @b score from theory, impl, deploy."""  
1396:         return (self.b_theory + self.b_impl + self.b_deploy) / 3.0  
1397:  
1398:  
1399: class IntrinsicCalibrationLoader:  
1400:     """
```

```

1401:     Single source loader for intrinsic calibration with fallback behavior.
1402:
1403:     Fallback rules:
1404:         - computed: Use actual values from JSON
1405:         - pending: b_theory=0.5, b_impl=0.5, b_deploy=0.5 (neutral baseline)
1406:         - excluded: Return None (signals method should be skipped)
1407:         - none: b_theory=0.3, b_impl=0.3, b_deploy=0.3 (low confidence + warning)
1408:     """
1409:
1410:    def __init__(self, config_path: str = "config/intrinsic_calibration.json") -> None:
1411:        self.config_path = Path(config_path)
1412:        self._data: dict = {}
1413:        self._load()
1414:
1415:    def _load(self) -> None:
1416:        """Load calibration data from JSON."""
1417:        if not self.config_path.exists():
1418:            raise FileNotFoundError(
1419:                f"Intrinsic calibration file not found: {self.config_path}. "
1420:                "Intrinsic calibration incomplete or contaminated."
1421:            )
1422:
1423:        with open(self.config_path) as f:
1424:            self._data = json.load(f)
1425:
1426:        metadata = self._data.get("_metadata", {})
1427:        coverage = metadata.get("coverage_percent", 0)
1428:
1429:        if coverage < 25.0:
1430:            raise ValueError(
1431:                f"Intrinsic calibration coverage {coverage}% < 25%. "
1432:                "Intrinsic calibration incomplete or contaminated."
1433:            )
1434:
1435:        logger.info(
1436:            f"Loaded intrinsic calibration: {metadata.get('computed_methods')} methods, "
1437:            f"{coverage}% coverage"
1438:        )
1439:
1440:    def get_calibration(self, method_id: str) -> IntrinsicCalibration | None:
1441:        """
1442:            Get intrinsic calibration for a method with fallback behavior.
1443:
1444:            Args:
1445:                method_id: Method identifier (e.g., "ClassName.method_name")
1446:
1447:            Returns:
1448:                IntrinsicCalibration object or None if method is excluded
1449:
1450:            Raises:
1451:                ValueError: If calibration data is contaminated
1452:        """
1453:
1454:        if method_id not in self._data:
1455:            logger.warning(
1456:                f"Method '{method_id}' not in calibration registry. "

```

```
1457:         )
1458:         return IntrinsicCalibration(
1459:             method_id=method_id,
1460:             intrinsic_score=(0.28, 0.32),
1461:             b_theory=0.3,
1462:             b_impl=0.3,
1463:             b_deploy=0.3,
1464:             calibration_status="none",
1465:             layer="utility",
1466:             last_updated="unknown"
1467:         )
1468:
1469:     method_data = self._data[method_id]
1470:
1471:     # Verify no contamination
1472:     allowed_keys = {"intrinsic_score", "b_theory", "b_impl", "b_deploy",
1473:                     "calibration_status", "layer", "last_updated"}
1474:     extra_keys = set(method_data.keys()) - allowed_keys
1475:     if extra_keys:
1476:         raise ValueError(
1477:             f"CONTAMINATION DETECTED in method '{method_id}': {extra_keys}. "
1478:             "Intrinsic calibration incomplete or contaminated."
1479:         )
1480:
1481:     status = method_data["calibration_status"]
1482:
1483:     # Handle fallback cases
1484:     if status == "excluded":
1485:         logger.info(f"Method '{method_id}' is excluded, returning None (skip method)")
1486:         return None
1487:
1488:     if status == "pending":
1489:         logger.info(f"Method '{method_id}' is pending, applying fallback @b=0.5")
1490:         return IntrinsicCalibration(
1491:             method_id=method_id,
1492:             intrinsic_score=(0.48, 0.52),
1493:             b_theory=0.5,
1494:             b_impl=0.5,
1495:             b_deploy=0.5,
1496:             calibration_status=status,
1497:             layer=method_data["layer"],
1498:             last_updated=method_data["last_updated"]
1499:         )
1500:
1501:     if status == "none":
1502:         logger.warning(
1503:             f"Method '{method_id}' has status='none', applying fallback @b=0.3"
1504:         )
1505:     return IntrinsicCalibration(
1506:         method_id=method_id,
1507:         intrinsic_score=(0.28, 0.32),
1508:         b_theory=0.3,
1509:         b_impl=0.3,
1510:         b_deploy=0.3,
1511:         calibration_status=status,
1512:         layer=method_data["layer"],
```

```
1513:             last_updated=method_data["last_updated"]
1514:         )
1515:
1516:     # status == "computed": return actual values
1517:     return IntrinsicCalibration(
1518:         method_id=method_id,
1519:         intrinsic_score=tuple(method_data["intrinsic_score"]),
1520:         b_theory=method_data["b_theory"],
1521:         b_impl=method_data["b_impl"],
1522:         b_deploy=method_data["b_deploy"],
1523:         calibration_status=status,
1524:         layer=method_data["layer"],
1525:         last_updated=method_data["last_updated"]
1526:     )
1527:
1528:     def get_metadata(self) -> dict:
1529:         """Get calibration metadata."""
1530:         return self._data.get("_metadata", {})
1531:
1532:     def verify_purity(self) -> bool:
1533:         """
1534:             Verify no contamination from other calibration layers.
1535:
1536:             Returns:
1537:                 True if pure @b-only data
1538:
1539:             Raises:
1540:                 ValueError: If contamination detected
1541:         """
1542:         forbidden_patterns = ["@chain", "@q", "@d", "@p", "@C", "@u", "@m",
1543:                               "final_score", "layer_scores", "chain_", "queue_"]
1544:
1545:         for method_id, method_data in self._data.items():
1546:             if method_id == "_metadata":
1547:                 continue
1548:
1549:             for key in method_data:
1550:                 for pattern in forbidden_patterns:
1551:                     if pattern in key.lower():
1552:                         raise ValueError(
1553:                             f"CONTAMINATION DETECTED: method '{method_id}' contains "
1554:                             f"forbidden key '{key}' matching pattern '{pattern}'. "
1555:                             "Intrinsic calibration incomplete or contaminated."
1556:                         )
1557:
1558:             return True
1559:
1560:
1561: # Singleton instance
1562: _loader: IntrinsicCalibrationLoader | None = None
1563:
1564:
1565: def get_intrinsic_calibration_loader(
1566:     config_path: str = "config/intrinsic_calibration.json"
1567: ) -> IntrinsicCalibrationLoader:
1568:     """Get singleton instance of intrinsic calibration loader."""
```

```
1569:     global _loader
1570:     if _loader is None:
1571:         _loader = IntrinsicCalibrationLoader(config_path)
1572:         _loader.verify_purity()
1573:     return _loader
1574:
1575:
1576: def get_method_calibration(method_id: str) -> IntrinsicCalibration | None:
1577:     """
1578:     Convenience function to get calibration for a method.
1579:
1580:     Args:
1581:         method_id: Method identifier (e.g., "ClassName.method_name")
1582:
1583:     Returns:
1584:         IntrinsicCalibration object or None if excluded
1585:     """
1586:     loader = get_intrinsic_calibration_loader()
1587:     return loader.get_calibration(method_id)
1588:
1589:
1590:
1591: =====
1592: FILE: src/farfan_pipeline/core/calibration/layer_assignment.py
1593: =====
1594:
1595: """
1596: Layer Assignment System for Calibration
1597:
1598: This module defines the canonical layer requirements for all method roles
1599: and provides layer assignment with Choquet integral weights for executors.
1600:
1601: Layers:
1602: - @b: Code quality (base theory)
1603: - @chain: Method wiring/orchestration
1604: - @q: Question appropriateness
1605: - @d: Dimension alignment
1606: - @p: Policy area fit
1607: - @C: Contract compliance
1608: - @u: Document quality
1609: - @m: Governance maturity
1610: """
1611:
1612: import re
1613: from typing import Any
1614:
1615: LAYER_REQUIREMENTS: dict[str, list[str]] = {
1616:     "ingest": ["@b", "@chain", "@u", "@m"],
1617:     "processor": ["@b", "@chain", "@u", "@m"],
1618:     "analyzer": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
1619:     "score": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
1620:     "executor": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
1621:     "utility": ["@b", "@chain", "@m"],
1622:     "orchestrator": ["@b", "@chain", "@m"],
1623:     "core": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
1624:     "extractor": ["@b", "@chain", "@u", "@m"],
```

```
1625: }
1626:
1627: CHOQUET_WEIGHTS: dict[str, float] = {
1628:     "@b": 0.17,
1629:     "@chain": 0.13,
1630:     "@q": 0.08,
1631:     "@d": 0.07,
1632:     "@p": 0.06,
1633:     "@C": 0.08,
1634:     "@u": 0.04,
1635:     "@m": 0.04,
1636: }
1637:
1638: CHOQUET_INTERACTION_WEIGHTS: dict[tuple[str, str], float] = {
1639:     ("@u", "@chain"): 0.13,
1640:     ("@chain", "@C"): 0.10,
1641:     ("@q", "@d"): 0.10,
1642: }
1643:
1644:
1645: def identify_executors(executors_file_path: str) -> list[dict[str, Any]]:
1646:     """
1647:         Identify all D[1-6]Q[1-5] executors from the executors.py file.
1648:
1649:     Args:
1650:         executors_file_path: Path to executors.py
1651:
1652:     Returns:
1653:         List of executor metadata dicts with method_id, role, dimension, question
1654:
1655:     Raises:
1656:         RuntimeError: If <30 executors found
1657:     """
1658:     with open(executors_file_path) as f:
1659:         content = f.read()
1660:
1661:     pattern = re.compile(r"class (D([1-6])_Q([1-5])_\w+)\(")
1662:     matches = pattern.findall(content)
1663:
1664:     executors = []
1665:     for class_name, dim, question in matches:
1666:         method_id = f"farfan_pipeline.core.orchestrator.executors.{class_name}"
1667:         executors.append(
1668:             {
1669:                 "method_id": method_id,
1670:                 "class_name": class_name,
1671:                 "dimension": f"D{dim}",
1672:                 "question": f"Q{question}",
1673:                 "role": "executor",
1674:                 "type": "analyzer",
1675:             }
1676:         )
1677:
1678:     if len(executors) < 30:
1679:         raise RuntimeError(
1680:             f"layer assignment corrupted: Found {len(executors)} executors, expected 30"
```

```
1681:         )
1682:
1683:     return executors
1684:
1685:
1686: def assign_layers_and_weights(
1687:     method_id: str, role: str, dimension: str = None, question: str = None
1688: ) -> dict[str, Any]:
1689:     """
1690:         Assign layers and Choquet weights to a method based on its role.
1691:
1692:     Args:
1693:         method_id: Fully qualified method identifier
1694:         role: Method role (ingest, processor, analyzer, etc.)
1695:         dimension: Dimension ID (D1-D6) for executors
1696:         question: Question ID (Q1-Q5) for executors
1697:
1698:     Returns:
1699:         Dict with layers, weights, and aggregator_type
1700:
1701:     Raises:
1702:         ValueError: If role not found in LAYER_REQUIREMENTS
1703:     """
1704:     if role not in LAYER_REQUIREMENTS:
1705:         raise ValueError(f"Unknown role: {role}")
1706:
1707:     layers = LAYER_REQUIREMENTS[role]
1708:
1709:     weights = {layer: CHOQUET_WEIGHTS[layer] for layer in layers}
1710:
1711:     interaction_weights = {}
1712:     for (l1, l2), weight in CHOQUET_INTERACTION_WEIGHTS.items():
1713:         if l1 in layers and l2 in layers:
1714:             interaction_weights[f"{l1},{l2}"] = weight
1715:
1716:     sum_linear = sum(weights.values())
1717:     sum_interaction = sum(interaction_weights.values())
1718:     total = sum_linear + sum_interaction
1719:
1720:     if abs(total - 1.0) > 0.01:
1721:         scale = 1.0 / total
1722:         weights = {k: v * scale for k, v in weights.items()}
1723:         interaction_weights = {k: v * scale for k, v in interaction_weights.items()}
1724:
1725:     return {
1726:         "method_id": method_id,
1727:         "role": role,
1728:         "dimension": dimension,
1729:         "question": question,
1730:         "layers": layers,
1731:         "weights": weights,
1732:         "interaction_weights": interaction_weights,
1733:         "aggregator_type": "choquet",
1734:     }
1735:
1736:
```

```
1737: def generate_canonical_inventory(executors_file_path: str) -> dict[str, Any]:
1738:     """
1739:         Generate the canonical inventory of methods with layer assignments.
1740:
1741:     Args:
1742:         executors_file_path: Path to executors.py
1743:
1744:     Returns:
1745:         Dict with metadata for all methods (NO SCORES)
1746:
1747:     Raises:
1748:         RuntimeError: If any validation fails
1749:     """
1750:     executors = identify_executors(executors_file_path)
1751:
1752:     inventory = {
1753:         "_metadata": {
1754:             "version": "1.0.0",
1755:             "description": "Canonical layer assignments for F.A.R.F.A.N. calibration system",
1756:             "total_executors": len(executors),
1757:             "layer_system": {
1758:                 "@b": "Code quality (base theory)",
1759:                 "@chain": "Method wiring/orchestration",
1760:                 "@q": "Question appropriateness",
1761:                 "@d": "Dimension alignment",
1762:                 "@p": "Policy area fit",
1763:                 "@c": "Contract compliance",
1764:                 "@u": "Document quality",
1765:                 "@m": "Governance maturity",
1766:             },
1767:             },
1768:             "methods": {},
1769:         }
1770:
1771:     for executor in executors:
1772:         assignment = assign_layers_and_weights(
1773:             method_id=executor["method_id"],
1774:             role=executor["role"],
1775:             dimension=executor["dimension"],
1776:             question=executor["question"],
1777:         )
1778:
1779:         if len(assignment["layers"]) < 8:
1780:             raise RuntimeError(
1781:                 f"layer assignment corrupted: Executor {executor['method_id']} "
1782:                 f"has {len(assignment['layers'])} layers, expected 8"
1783:             )
1784:
1785:         weights_sum = sum(assignment["weights"].values()) + sum(
1786:             assignment["interaction_weights"].values()
1787:         )
1788:         if abs(weights_sum - 1.0) > 0.01:
1789:             raise RuntimeError(
1790:                 f"layer assignment corrupted: Weights for {executor['method_id']} "
1791:                 f"sum to {weights_sum}, expected 1.0"
1792:             )
```

```
1793:
1794:     inventory["methods"][executor["method_id"]] = {
1795:         "method_id": assignment["method_id"],
1796:         "role": assignment["role"],
1797:         "dimension": assignment["dimension"],
1798:         "question": assignment["question"],
1799:         "layers": assignment["layers"],
1800:         "weights": assignment["weights"],
1801:         "interaction_weights": assignment["interaction_weights"],
1802:         "aggregator_type": assignment["aggregator_type"],
1803:     }
1804:
1805:     return inventory
1806:
1807:
1808:
1809: =====
1810: FILE: src/farfan_pipeline/core/calibration/layer_coexistence.py
1811: =====
1812:
1813: """
1814: Layer Coexistence and Influence Framework
1815:
1816: Formal mathematical framework for method calibration based on layer-specific
1817: evidence aggregation with theoretically grounded fusion operators.
1818:
1819: Theoretical Foundations:
1820: - Bayesian inference (Pearl, 1988; Jaynes, 2003)
1821: - Multi-criteria decision analysis (Keeney & Raiffa, 1976)
1822: - Policy coherence structures (Nilsson et al., 2012)
1823: - Ensemble learning (Dietterich, 2000; Wolpert, 1992)
1824: - Fuzzy measures and aggregation (Yager, 1988; Beliakov et al., 2007)
1825:
1826: Canonical Layer Notation (from questionnaire_monolith.json):
1827: - @q: Question Layer (300 questions: Q001-Q300)
1828: - @d: Dimension Layer (6 dimensions: DIM01-DIM06)
1829: - @p: Policy Area Layer (10 areas: PA01-PA10)
1830: - @C: Congruence Layer (ensemble compatibility)
1831: - @m: Meta Layer (cross-layer aggregation)
1832: """
1833:
1834: from dataclasses import dataclass, field
1835: from typing import Dict, List, Set, Optional
1836: from enum import Enum
1837:
1838:
1839: class Layer(Enum):
1840:     """
1841:     Canonical layer identifiers.
1842:
1843:     Source: questionnaire_monolith.json and theoretical framework specification.
1844:     These are the ONLY valid layer identifiers in the system.
1845:     """
1846:     QUESTION = "@q"      # Evidence-weighted Bayesian scoring
1847:     DIMENSION = "@d"    # Multi-criteria value functions
1848:     POLICY_AREA = "@p"  # Policy coherence structures
```

```
1849:     CONGRUENCE = "@C"      # Ensemble compatibility
1850:     META = "@m"          # Generalized aggregation
1851:
1852:
1853: @dataclass(frozen=True)
1854: class LayerScore:
1855:     """
1856:         Score from a single layer for a method.
1857:
1858:     Attributes:
1859:         layer: The layer identifier
1860:         value: Numerical score in [0, 1]
1861:         weight: Importance weight in [0, 1]
1862:         metadata: Additional layer-specific information
1863:     """
1864:     layer: Layer
1865:     value: float
1866:     weight: float = 1.0
1867:     metadata: Dict = field(default_factory=dict)
1868:
1869:     def __new__(cls, layer: Layer, value: float, weight: float = 1.0, metadata: Dict = None):
1870:         """Validate score bounds before instance creation"""
1871:         if not 0 <= value <= 1:
1872:             raise ValueError(f"Layer score must be in [0,1], got {value}")
1873:         if not 0 <= weight <= 1:
1874:             raise ValueError(f"Layer weight must be in [0,1], got {weight}")
1875:         if metadata is None:
1876:             metadata = {}
1877:         return super().__new__(cls)
1878:
1879:     def __init__(self, layer: Layer, value: float, weight: float = 1.0, metadata: Dict = None):
1880:         # dataclass will set fields, but we need to ensure metadata is not None
1881:         if metadata is None:
1882:             object.__setattr__(self, 'metadata', {})
1883: @dataclass
1884: class MethodSignature:
1885:     """
1886:         Complete signature for a method under Layer Coexistence framework.
1887:
1888:         This is the canonical method notation that every calibrated method must expose.
1889:
1890:     Attributes:
1891:         method_id: Unique identifier (ClassName.method_name)
1892:         active_layers: Set of layers relevant to this method (L(M))
1893:         input_schema: Dict describing required inputs
1894:         output_schema: Dict describing output space
1895:         fusion_operator_name: Name of the fusion operator F_M
1896:         fusion_parameters: Parameters for F_M
1897:         calibration_rule: Human-readable calibration rule description
1898:     """
1899:     method_id: str
1900:     active_layers: Set[Layer]
1901:     input_schema: Dict
1902:     output_schema: Dict
1903:     fusion_operator_name: str
1904:     fusion_parameters: Dict
```

```
1905:     calibration_rule: str
1906:
1907:     def to_dict(self) -> Dict:
1908:         """Export to dictionary for serialization"""
1909:         return {
1910:             'method_id': self.method_id,
1911:             'active_layers': [layer.value for layer in self.active_layers],
1912:             'input_schema': self.input_schema,
1913:             'output_schema': self.output_schema,
1914:             'fusion_operator_name': self.fusion_operator_name,
1915:             'fusion_parameters': self.fusion_parameters,
1916:             'calibration_rule': self.calibration_rule
1917:         }
1918:
1919:     @classmethod
1920:     def from_dict(cls, data: Dict) -> 'MethodSignature':
1921:         """Load from dictionary"""
1922:         return cls(
1923:             method_id=data['method_id'],
1924:             active_layers={Layer(layer_str) for layer_str in data['active_layers']},
1925:             input_schema=data['input_schema'],
1926:             output_schema=data['output_schema'],
1927:             fusion_operator_name=data['fusion_operator_name'],
1928:             fusion_parameters=data['fusion_parameters'],
1929:             calibration_rule=data['calibration_rule']
1930:         )
1931:
1932:
1933: class FusionOperator:
1934:     """
1935:     Abstract base class for fusion operators F_M.
1936:
1937:     All fusion operators must satisfy:
1938:     - Monotonicity:  $\forall i \in [0, n] \forall j \in [0, n] \text{ if } i < j \text{ then } f(i) \leq f(j)$ 
1939:     - Boundedness:  $f: [0, 1]^n \rightarrow [0, 1]$ 
1940:     - Interpretability: Clear semantic meaning of output
1941:
1942:     Subclasses must implement:
1943:     - fuse(scores: List[LayerScore]) -> float
1944:     - verify_properties() -> Dict[str, bool]
1945:     - get_formula() -> str
1946:     """
1947:
1948:     def __init__(self, name: str, parameters: Dict):
1949:         self.name = name
1950:         self.parameters = parameters
1951:
1952:     def fuse(self, scores: List[LayerScore]) -> float:
1953:         """
1954:             Aggregate layer scores into calibrated output.
1955:
1956:             Args:
1957:                 scores: List of LayerScore objects
1958:
1959:             Returns:
1960:                 Calibrated score in [0, 1]
```

```

1961: """
1962:     raise NotImplementedError("Subclasses must implement fuse()")
1963:
1964:     def verify_properties(self) -> Dict[str, bool]:
1965:         """
1966:             Verify mathematical properties (monotonicity, boundedness, etc.)
1967:
1968:             Returns:
1969:                 Dict mapping property name to verification result
1970:             """
1971:     raise NotImplementedError("Subclasses must implement verify_properties()")
1972:
1973:     def get_formula(self) -> str:
1974:         """
1975:             Return explicit mathematical formula in canonical notation.
1976:
1977:             Returns:
1978:                 LaTeX-style formula string
1979:             """
1980:     raise NotImplementedError("Subclasses must implement get_formula()")
1981:
1982:     def get_trace(self, scores: List[LayerScore]) -> List[str]:
1983:         """
1984:             Generate step-by-step arithmetic trace.
1985:
1986:             Args:
1987:                 scores: List of LayerScore objects
1988:
1989:             Returns:
1990:                 List of computation steps as strings
1991:             """
1992:     raise NotImplementedError("Subclasses must implement get_trace()")
1993:
1994:
1995: class WeightedAverageFusion(FusionOperator):
1996:     """
1997:         Weighted average fusion operator.
1998:
1999:         Formula:  $F_M(x) = \frac{\sum w_i x_i}{\sum w_i}$ 
2000:
2001:         Properties:
2002:             - Monotonic: Yes ( $\forall x_1, x_2 \in X, \forall w_1, w_2 \in W, w_1 > w_2 \Rightarrow F_M(x_1, w_1) > F_M(x_2, w_2)$ )
2003:             - Bounded: Yes ( $\min(x) \leq F_M(x) \leq \max(x)$ )
2004:             - Idempotent: Yes ( $F_M(c, c, \dots, c) = c$ )
2005:             - Compensatory: Full (low scores can be compensated by high scores)
2006:
2007:         Reference: Standard weighted mean in MCDA (Keeney & Raiffa, 1976, Ch. 3)
2008:     """
2009:
2010:     def __init__(self, parameters: Optional[Dict] = None):
2011:         super().__init__("WeightedAverage", parameters or {})
2012:         self.normalize_weights = parameters.get('normalize_weights', True) if parameters else True
2013:
2014:     def fuse(self, scores: List[LayerScore]) -> float:
2015:         """Compute weighted average"""
2016:         if not scores:

```

```

2017:         return 0.0
2018:
2019:     weighted_sum = sum(score.value * score.weight for score in scores)
2020:     weight_sum = sum(score.weight for score in scores)
2021:
2022:     if weight_sum == 0:
2023:         return 0.0
2024:
2025:     return weighted_sum / weight_sum
2026:
2027: def verify_properties(self) -> Dict[str, bool]:
2028:     """Verify mathematical properties"""
2029:     # Test monotonicity with sample inputs
2030:     test_scores_low = [LayerScore(Layer.QUESTION, 0.3, 1.0)]
2031:     test_scores_high = [LayerScore(Layer.QUESTION, 0.7, 1.0)]
2032:
2033:     result_low = self.fuse(test_scores_low)
2034:     result_high = self.fuse(test_scores_high)
2035:
2036:     return {
2037:         'monotonic': result_high >= result_low,
2038:         'bounded': 0 <= result_low <= 1 and 0 <= result_high <= 1,
2039:         'idempotent': abs(self.fuse([LayerScore(Layer.QUESTION, 0.5, 1.0)]) - 0.5) < 1e-10
2040:     }
2041:
2042: def get_formula(self) -> str:
2043:     """Return LaTeX formula"""
2044:     return r"F_{WA}(x) = \frac{\sum_{\ell \in L(M)} w_\ell \cdot x_\ell}{\sum_{\ell \in L(M)} w_\ell}"
2045:
2046: def get_trace(self, scores: List[LayerScore]) -> List[str]:
2047:     """Generate computation trace"""
2048:     trace = []
2049:     trace.append(f"Weighted Average Fusion: {len(scores)} layers")
2050:
2051:     for i, score in enumerate(scores):
2052:         trace.append(f"  Layer {score.layer.value}: x = {score.value:.4f}, w = {score.weight:.4f}")
2053:
2054:     weighted_sum = sum(s.value * s.weight for s in scores)
2055:     weight_sum = sum(s.weight for s in scores)
2056:
2057:     trace.append(f"Weighted sum: {weighted_sum:.4f}")
2058:     trace.append(f"Weight sum: {weight_sum:.4f}")
2059:     if weight_sum == 0:
2060:         trace.append(f"Result: No valid weights, returning 0.0")
2061:     else:
2062:         trace.append(f"Result: {weighted_sum:.4f} / {weight_sum:.4f} = {weighted_sum/weight_sum:.4f}")
2063:
2064:     return trace
2065:
2066:
2067: class OWAOperator(FusionOperator):
2068:     """
2069:     Ordered Weighted Averaging (OWA) fusion operator.
2070:
2071:     Formula: F_OWA(x) = \sum_i v_i \cdot x_{(i)}
2072:     where x_{(i)} is the i-th largest value and v_i are position weights

```

```

2073:
2074:     Properties:
2075:         - Monotonic: Yes (if all v_i >= 0)
2076:         - Bounded: Yes
2077:         - Allows modeling of optimism/pessimism (andness/orness)
2078:
2079:     Reference: Yager (1988) "On ordered weighted averaging aggregation operators"
2080:                 Int. J. General Systems, 14(3), 183-194
2081:
2082:     Parameters:
2083:         weights: Position-based weights [v_1, v_2, ..., v_n]
2084:             Should sum to 1 for proper normalization
2085:
2086:
2087:     def __init__(self, parameters: Dict):
2088:         super().__init__("OWA", parameters)
2089:         self.position_weights = parameters.get('weights', [])
2090:
2091:         if len(self.position_weights) == 0:
2092:             raise ValueError("OWA requires position weights")
2093:
2094:     def fuse(self, scores: List[LayerScore]) -> float:
2095:         """Compute OWA aggregation"""
2096:         if not scores:
2097:             return 0.0
2098:
2099:         # Sort scores in descending order
2100:         values = [score.value for score in scores]
2101:         sorted_values = sorted(values, reverse=True)
2102:
2103:         # Pad or truncate weights if necessary
2104:         n = len(sorted_values)
2105:         if len(self.position_weights) < n:
2106:             # Extend with equal weights
2107:             extended_weights = list(self.position_weights) + [1.0/n] * (n - len(self.position_weights))
2108:         else:
2109:             extended_weights = self.position_weights[:n]
2110:
2111:         # Normalize weights
2112:         weight_sum = sum(extended_weights)
2113:         if weight_sum > 0:
2114:             extended_weights = [w / weight_sum for w in extended_weights]
2115:
2116:         # Compute weighted sum
2117:         result = sum(w * v for w, v in zip(extended_weights, sorted_values))
2118:         return float(result)
2119:
2120:     def verify_properties(self) -> Dict[str, bool]:
2121:         """Verify OWA properties"""
2122:         # Check monotonicity, boundedness
2123:         test_scores = [
2124:             LayerScore(Layer.QUESTION, 0.2, 1.0),
2125:             LayerScore(Layer.DIMENSION, 0.5, 1.0),
2126:             LayerScore(Layer.POLICY_AREA, 0.8, 1.0)
2127:         ]
2128:
```

```

2129:         result = self.fuse(test_scores)
2130:         weight_sum = sum(self.position_weights)
2131:
2132:     return {
2133:         'monotonic': True, # Always true if weights are non-negative
2134:         'bounded': 0 <= result <= 1,
2135:         'weights_sum_to_one': abs(weight_sum - 1.0) < 1e-6
2136:     }
2137:
2138: def get_formula(self) -> str:
2139:     """Return LaTeX formula"""
2140:     return r"F_{OWA}(x) = \sum_{i=1}^n v_i \cdot x_{(i)} \text{ where } x_{(i)} \text{ is i-th largest}"
2141:
2142: def get_trace(self, scores: List[LayerScore]) -> List[str]:
2143:     """Generate computation trace"""
2144:     trace = []
2145:     trace.append(f"OWA Fusion: {len(scores)} layers")
2146:
2147:     values = [(score.layer.value, score.value) for score in scores]
2148:     values_sorted = sorted(values, key=lambda x: x[1], reverse=True)
2149:
2150:     trace.append("Sorted values (descending):")
2151:     for i, (layer, val) in enumerate(values_sorted):
2152:         weight_idx = min(i, len(self.position_weights) - 1)
2153:         weight = self.position_weights[weight_idx]
2154:         trace.append(f" Position {i+1}: {layer} = {val:.4f}, weight = {weight:.4f}")
2155:
2156:     result = self.fuse(scores)
2157:     trace.append(f"Result: {result:.4f}")
2158:
2159:     return trace
2160:
2161:
2162: # Registry of available fusion operators
2163: FUSION_OPERATORS = {
2164:     'WeightedAverage': WeightedAverageFusion,
2165:     'OWA': OWA,
2166: }
2167:
2168:
2169: def create_fusion_operator(name: str, parameters: Optional[Dict] = None) -> FusionOperator:
2170:     """
2171:         Factory function to create fusion operators.
2172:
2173:     Args:
2174:         name: Operator name from FUSION_OPERATORS
2175:         parameters: Operator-specific parameters
2176:
2177:     Returns:
2178:         Configured FusionOperator instance
2179:     """
2180:     if name not in FUSION_OPERATORS:
2181:         raise ValueError(f"Unknown fusion operator: {name}. Available: {list(FUSION_OPERATORS.keys())}")
2182:
2183:     operator_class = FUSION_OPERATORS[name]
2184:     return operator_class(parameters or {})

```

```
2185:  
2186:  
2187:  
2188: =====  
2189: FILE: src/farfan_pipeline/core/calibration/layer_computers.py  
2190: =====  
2191:  
2192: """  
2193: Three-Pillar Calibration System - Layer Computation Functions  
2194:  
2195: This module implements the 8 layer score computation functions as specified  
2196: in the SUPERPROMPT Three-Pillar Calibration System.  
2197:  
2198: Spec compliance: Section 3 (Layer Architecture)  
2199: """  
2200:  
2201: import json  
2202: import math  
2203: from pathlib import Path  
2204: from typing import Any  
2205:  
2206: from farfan_pipeline.core.calibration.data_structures import (  
2207:     CalibrationConfigError,  
2208:     ComputationGraph,  
2209:     InterplaySubgraph,  
2210:     MethodRole,  
2211: )  
2212:  
2213: _unit_transforms_config: dict[str, Any] | None = None  
2214:  
2215:  
2216: def _load_unit_transforms_config() -> dict[str, Any]:  
2217:     """Load unit transforms configuration from system/config/calibration/unit_transforms.json."""  
2218:     global _unit_transforms_config  
2219:  
2220:     if _unit_transforms_config is not None:  
2221:         return _unit_transforms_config  
2222:  
2223:     config_path = Path("system/config/calibration/unit_transforms.json")  
2224:     if not config_path.exists():  
2225:         raise FileNotFoundError(f"Unit transforms config not found: {config_path}")  
2226:  
2227:     with open(config_path, encoding='utf-8') as f:  
2228:         _unit_transforms_config = json.load(f)  
2229:  
2230:     return _unit_transforms_config  
2231:  
2232:  
2233: def compute_base_layer(method_id: str, intrinsic_config: dict[str, Any]) -> float:  
2234:     """  
2235:     Compute base layer score (@b): Intrinsic quality  
2236:  
2237:     Spec compliance: Section 3.1  
2238:     Formula: x_@b = w_th * b_theory + w_imp * b_impl + w_dep * b_deploy  
2239:  
2240:     Args:
```

```
2241:     method_id: Canonical method ID
2242:     intrinsic_config: Loaded intrinsic_calibration.json
2243:
2244:     Returns:
2245:         Score in [0,1]
2246:
2247:     Raises:
2248:         ValueError: If method not found or scores invalid
2249:     """
2250:     if method_id not in intrinsic_config.get("methods", {}):
2251:         raise ValueError(f"Method {method_id} not found in intrinsic_calibration.json")
2252:
2253:     method_data = intrinsic_config["methods"][method_id]
2254:     weights = intrinsic_config["_base_weights"]
2255:
2256:     b_theory = method_data["b_theory"]
2257:     b_impl = method_data["b_impl"]
2258:     b_deploy = method_data["b_deploy"]
2259:
2260:     # Validate bounds
2261:     for name, value in [("b_theory", b_theory), ("b_impl", b_impl), ("b_deploy", b_deploy)]:
2262:         if not (0.0 <= value <= 1.0):
2263:             raise ValueError(f"{name} must be in [0,1], got {value}")
2264:
2265:     # Compute weighted sum
2266:     score = (weights["w_th"] * b_theory +
2267:              weights["w_imp"] * b_impl +
2268:              weights["w_dep"] * b_deploy)
2269:
2270:     return score
2271:
2272:
2273: def compute_chain_layer(node_id: str, graph: ComputationGraph,
2274:                         contextual_config: dict[str, Any]) -> float:
2275:     """
2276:     Compute chain compatibility layer (@chain)
2277:
2278:     Spec compliance: Section 3.2
2279:     Rule-based discrete mapping
2280:
2281:     Args:
2282:         node_id: Node identifier
2283:         graph: Computation graph containing node
2284:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)
2285:
2286:     Returns:
2287:         Score in [0,1]
2288:     """
2289:     if node_id not in graph.nodes:
2290:         raise ValueError(f"Node {node_id} not in graph")
2291:
2292:     config = _load_unit_transforms_config()
2293:     mappings = config.get("chain_layer", {}).get("discrete_mappings", {})
2294:
2295:     if not mappings:
2296:         mappings = contextual_config.get("layer_chain", {}).get("discrete_mappings", {})
```

```
2297:  
2298:     signature = graph.node_signatures.get(node_id, {})  
2299:     required_inputs = signature.get("required_inputs", [])  
2300:  
2301:     has_hard_mismatch = False  
2302:  
2303:     incoming_edges = [e for e in graph.edges if e[1] == node_id]  
2304:  
2305:     if not incoming_edges and required_inputs:  
2306:         has_hard_mismatch = True  
2307:  
2308:     if has_hard_mismatch:  
2309:         return mappings.get("hard_mismatch", 0.0)  
2310:     else:  
2311:         return mappings.get("all_contracts_pass_no_warnings", 1.0)  
2312:  
2313:  
2314: def compute_unit_layer(method_id: str, role: MethodRole, unit_quality: float,  
2315:                         contextual_config: dict[str, Any]) -> float:  
2316:     """  
2317:     Compute unit-of-analysis sensitivity layer (@u)  
2318:  
2319:     Spec compliance: Section 3.3  
2320:     Formula: x_@u = g_M(U) if M is U-sensitive, else 1.0  
2321:  
2322:     Args:  
2323:         method_id: Canonical method ID  
2324:         role: Method role  
2325:         unit_quality: U in [0,1]  
2326:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)  
2327:  
2328:     Returns:  
2329:         Score in [0,1]  
2330:     """  
2331:     if not (0.0 <= unit_quality <= 1.0):  
2332:         raise ValueError(f"unit_quality must be in [0,1], got {unit_quality}")  
2333:  
2334:     config = _load_unit_transforms_config()  
2335:     g_functions_config = config.get("g_functions", {})  
2336:  
2337:     role_name = role.value  
2338:  
2339:     g_spec = None  
2340:     for g_name, g_def in g_functions_config.items():  
2341:         if role_name in g_def.get("applicable_roles", []):  
2342:             g_spec = g_def  
2343:             break  
2344:  
2345:     if g_spec is None:  
2346:         return 1.0  
2347:  
2348:     g_type = g_spec["type"]  
2349:  
2350:     if g_type == "identity":  
2351:         return unit_quality  
2352:
```

```

2353:     elif g_type == "constant":
2354:         return g_spec.get("value", 1.0)
2355:
2356:     elif g_type == "piecewise_linear":
2357:         abort_threshold = g_spec.get("abort_threshold", 0.3)
2358:         if unit_quality < abort_threshold:
2359:             return 0.0
2360:         slope = g_spec.get("slope", 2.0)
2361:         offset = g_spec.get("offset", -0.6)
2362:         score = slope * unit_quality + offset
2363:
2364:         if score < 0.0 or score > 1.0:
2365:             raise CalibrationConfigError(
2366:                 f"Unit layer g_function produced out-of-range score: {score} "
2367:                 f"for unit_quality={unit_quality}. Config must be adjusted to ensure [0,1] output."
2368:             )
2369:         return score
2370:
2371:     elif g_type == "sigmoidal":
2372:         k = g_spec.get("k", 5.0)
2373:         x0 = g_spec.get("x0", 0.5)
2374:         score = 1.0 - math.exp(-k * (unit_quality - x0))
2375:
2376:         if score < 0.0 or score > 1.0:
2377:             raise CalibrationConfigError(
2378:                 f"Unit layer g_function produced out-of-range score: {score} "
2379:                 f"for unit_quality={unit_quality}, k={k}, x0={x0}. "
2380:                 f"Config must be adjusted to ensure [0,1] output."
2381:             )
2382:         return score
2383:
2384:     else:
2385:         raise ValueError(f"Unknown g_function type: {g_type}")
2386:
2387:
2388: def compute_question_layer(method_id: str, question_id: str | None,
2389:                             monolith: dict[str, Any],
2390:                             contextual_config: dict[str, Any]) -> float:
2391: """
2392: Compute question compatibility layer (@q)
2393:
2394: Spec compliance: Section 3.4
2395: Formula: x@q = Q_f(M | Q)
2396:
2397: Args:
2398:     method_id: Canonical method ID
2399:     question_id: Question ID (or None)
2400:     monolith: Loaded questionnaire_monolith.json
2401:     contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)
2402:
2403: Returns:
2404:     Score in [0,1]
2405: """
2406: config = _load_unit_transforms_config()
2407: levels = config.get("question_layer", {}).get("compatibility_levels", {})
2408:
```

```
2409:     if not levels:
2410:         levels = contextual_config.get("layer_question", {}).get("compatibility_levels", {})
2411:
2412:     if question_id is None:
2413:         return levels.get("undeclared", 0.6)
2414:
2415:     micro_questions = monolith.get("blocks", {}).get("micro_questions", [])
2416:     question = None
2417:     for q in micro_questions:
2418:         if q.get("question_id") == question_id:
2419:             question = q
2420:             break
2421:
2422:     if not question:
2423:         return levels.get("undeclared", 0.6)
2424:
2425:     method_sets = question.get("method_sets", [])
2426:
2427:     for method_spec in method_sets:
2428:         if (method_id.endswith(f".{method_spec.get('function', '')}")) or
2429:             method_spec.get('class', '') in method_id:
2430:
2431:             method_type = method_spec.get("method_type", "")
2432:             priority = method_spec.get("priority", 99)
2433:
2434:             if method_type == "extraction" or priority == 1:
2435:                 return levels.get("primary", 1.0)
2436:             elif priority == 2:
2437:                 return levels.get("secondary", 0.8)
2438:             elif method_type == "validation":
2439:                 return levels.get("validator", 0.9)
2440:
2441:     return levels.get("undeclared", 0.6)
2442:
2443:
2444: def compute_dimension_layer(method_id: str, dimension_id: str,
2445:                             contextual_config: dict[str, Any]) -> float:
2446:     """
2447:     Compute dimension compatibility layer (@d)
2448:
2449:     Spec compliance: Section 3.5
2450:     Formula: x_@d = D_f(M | D)
2451:
2452:     Args:
2453:         method_id: Canonical method ID
2454:         dimension_id: Dimension ID (DIM01-DIM06)
2455:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)
2456:
2457:     Returns:
2458:         Score in [0,1]
2459:     """
2460:     config = _load_unit_transforms_config()
2461:     alignment = config.get("dimension_layer", {}).get("alignment_matrix", {})
2462:
2463:     if not alignment:
2464:         alignment = contextual_config.get("layer_dimension", {}).get("alignment_matrix", {})
```

```
2465:
2466:     if dimension_id not in alignment:
2467:         raise ValueError(f"Unknown dimension: {dimension_id}")
2468:
2469:     dim_spec = alignment[dimension_id]
2470:
2471:     return dim_spec.get("default_score", 1.0)
2472:
2473:
2474: def compute_policy_layer(method_id: str, policy_id: str,
2475:                         contextual_config: dict[str, Any]) -> float:
2476:     """
2477:     Compute policy area compatibility layer (@p)
2478:
2479:     Spec compliance: Section 3.6
2480:     Formula:  $x_{@p} = P_f(M \mid P)$ 
2481:
2482:     Args:
2483:         method_id: Canonical method ID
2484:         policy_id: Policy area ID (PA01-PA10)
2485:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)
2486:
2487:     Returns:
2488:         Score in [0,1]
2489:     """
2490:     config = _load_unit_transforms_config()
2491:     policies = config.get("policy_layer", {}).get("policy_areas", {})
2492:
2493:     if not policies:
2494:         policies = contextual_config.get("layer_policy", {}).get("policy_areas", {})
2495:
2496:     if policy_id not in policies:
2497:         raise ValueError(f"Unknown policy area: {policy_id}")
2498:
2499:     policy_spec = policies[policy_id]
2500:
2501:     return policy_spec.get("default_score", 0.9)
2502:
2503:
2504: def compute_interplay_layer(interplay: InterplaySubgraph | None,
2505:                            contextual_config: dict[str, Any]) -> float:
2506:     """
2507:     Compute interplay congruence layer (@C)
2508:
2509:     Spec compliance: Section 3.7
2510:     Formula:  $C_{play}(G \mid ctx) = c_{scale} \cdot c_{sem} \cdot c_{fusion}$ 
2511:
2512:     Args:
2513:         interplay: Interplay subgraph (or None)
2514:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)
2515:
2516:     Returns:
2517:         Score in [0,1]
2518:     """
2519:     config = _load_unit_transforms_config()
2520:     interplay_config = config.get("interplay_layer", {})
```

```
2521:  
2522:     if not interplay_config:  
2523:         interplay_config = contextual_config.get("layer_interplay", {})  
2524:  
2525:     if interplay is None:  
2526:         return interplay_config.get("default_when_not_in_interplay", 1.0)  
2527:  
2528:     components = interplay_config.get("components", {})  
2529:  
2530:     c_scale = components.get("c_scale", {}).get("same_range", 1.0)  
2531:     c_sem = 1.0  
2532:     c_fusion = components.get("c_fusion", {}).get("declared_and_satisfied", 1.0)  
2533:  
2534:     return c_scale * c_sem * c_fusion  
2535:  
2536:  
2537: def compute_meta_layer(evidence: dict[str, Any],  
2538:                         contextual_config: dict[str, Any]) -> float:  
2539:     """  
2540:     Compute meta/governance layer (@m)  
2541:  
2542:     Spec compliance: Section 3.8  
2543:     Formula:  $x_{@m} = 0.5 \cdot m_{transp} + 0.4 \cdot m_{gov} + 0.1 \cdot m_{cost}$   
2544:  
2545:     Args:  
2546:         evidence: Evidence dictionary with metrics  
2547:         contextual_config: Loaded contextual_parametrization.json (deprecated, use unit_transforms.json)  
2548:  
2549:     Returns:  
2550:         Score in [0,1]  
2551:     """  
2552:     config = _load_unit_transforms_config()  
2553:     meta_spec = config.get("meta_layer", {})  
2554:  
2555:     if not meta_spec:  
2556:         meta_spec = contextual_config.get("layer_meta", {})  
2557:  
2558:     transp_conditions = [  
2559:         evidence.get("formula_export_valid", False),  
2560:         evidence.get("trace_complete", False),  
2561:         evidence.get("logs_conform_schema", False)  
2562:     ]  
2563:     transp_count = sum(transp_conditions)  
2564:  
2565:     transp_values = meta_spec.get("components", {}).get("m_transp", {})  
2566:     if transp_count == 3:  
2567:         m_transp = transp_values.get("all_three_conditions", 1.0)  
2568:     elif transp_count == 2:  
2569:         m_transp = transp_values.get("two_of_three", 0.8)  
2570:     elif transp_count == 1:  
2571:         m_transp = transp_values.get("one_of_three", 0.5)  
2572:     else:  
2573:         m_transp = transp_values.get("none", 0.0)  
2574:  
2575:     gov_conditions = [  
2576:         evidence.get("version_tagged", False),
```

```

2577:         evidence.get("config_hash_matches", False),
2578:         evidence.get("signature_valid", False)
2579:     ]
2580:     gov_count = sum(gov_conditions)
2581:
2582:     gov_values = meta_spec.get("components", {}).get("m_gov", {})
2583:     if gov_count == 3:
2584:         m_gov = gov_values.get("all_three_conditions", 1.0)
2585:     elif gov_count == 2:
2586:         m_gov = gov_values.get("two_of_three", 0.8)
2587:     elif gov_count == 1:
2588:         m_gov = gov_values.get("one_of_three", 0.5)
2589:     else:
2590:         m_gov = gov_values.get("none", 0.0)
2591:
2592:     runtime_ms = evidence.get("runtime_ms", 100)
2593:     thresholds = meta_spec.get("components", {}).get("m_cost", {}).get("thresholds", {})
2594:
2595:     fast_threshold = thresholds.get("fast_runtime_ms", 50)
2596:     acceptable_threshold = thresholds.get("acceptable_runtime_ms", 200)
2597:
2598:     cost_values = meta_spec.get("components", {}).get("m_cost", {})
2599:     if runtime_ms < fast_threshold:
2600:         m_cost = cost_values.get("fast", 1.0)
2601:     elif runtime_ms < acceptable_threshold:
2602:         m_cost = cost_values.get("acceptable", 0.8)
2603:     else:
2604:         m_cost = cost_values.get("slow", 0.5)
2605:
2606:     weights = meta_spec.get("aggregation", {}).get("weights", {})
2607:     score = (weights.get("transparency", 0.5) * m_transp +
2608:             weights.get("governance", 0.4) * m_gov +
2609:             weights.get("cost", 0.1) * m_cost)
2610:
2611:     return score
2612:
2613:
2614:
2615: =====
2616: FILE: src/farfan_pipeline/core/calibration/layer_influence_model.py
2617: =====
2618:
2619: """
2620: Layer Coexistence and Influence Model - Formal Specification
2621:
2622: This module encodes the mathematical relationships between layers,
2623: including:
2624: - Conditional activation rules (when a layer becomes relevant)
2625: - Influence relationships (how layers weight/transform each other)
2626: - Coexistence constraints (compatibility requirements)
2627:
2628: All rules are explicit, verifiable, and derived from theoretical foundations.
2629:
2630: References:
2631: - Pearl (1988): Probabilistic Reasoning in Intelligent Systems (conditional independence)
2632: - Keeney & Raiffa (1976): Decisions with Multiple Objectives (preference independence)

```

```
2633: - Grabisch (1997): k-order additive discrete fuzzy measures (interaction indices)
2634: """
2635:
2636: from dataclasses import dataclass, field
2637: from typing import Dict, List, Set, Optional, Callable, Tuple
2638: from enum import Enum
2639: import json
2640:
2641: from farfan_pipeline.core.calibration.layer_coexistence import Layer, LayerScore
2642:
2643:
2644: class LayerInfluenceType(Enum):
2645:     """Types of influence one layer can have on another."""
2646:     WEIGHTING = "weighting"          # Layer A modifies weight of Layer B
2647:     TRANSFORMATION = "transformation" # Layer A transforms values from Layer B
2648:     ACTIVATION = "activation"        # Layer A determines if Layer B is active
2649:     CONSTRAINT = "constraint"       # Layer A constrains valid values of Layer B
2650:
2651:
2652: @dataclass
2653: class LayerInfluence:
2654:     """
2655:         Formal specification of how one layer influences another.
2656:
2657:         Attributes:
2658:             source_layer: The influencing layer
2659:             target_layer: The influenced layer
2660:             influence_type: Type of influence relationship
2661:             strength: Strength of influence in [0, 1]
2662:             functional_form: Mathematical description of influence
2663:             conditions: When this influence applies
2664:     """
2665:     source_layer: Layer
2666:     target_layer: Layer
2667:     influence_type: LayerInfluenceType
2668:     strength: float
2669:     functional_form: str
2670:     conditions: Dict = field(default_factory=dict)
2671:
2672:     def __post_init__(self):
2673:         """Validate influence specification"""
2674:         if not 0 <= self.strength <= 1:
2675:             raise ValueError(f"Influence strength must be in [0,1], got {self.strength}")
2676:         if self.source_layer == self.target_layer:
2677:             raise ValueError("Self-influence not permitted in current model")
2678:
2679:
2680: @dataclass
2681: class LayerActivationRule:
2682:     """
2683:         Rule determining when a layer becomes active for a method.
2684:
2685:         This encodes the endogenous determination of L(M) from method characteristics.
2686:
2687:         Attributes:
2688:             layer: The layer this rule applies to
```

```
2689:     triggers: Conditions that activate this layer
2690:     prerequisites: Other layers that must be active first
2691:     priority: Activation priority (higher = checked first)
2692: """
2693:     layer: Layer
2694:     triggers: List[Callable] # Functions that return bool
2695:     prerequisites: Set[Layer] = field(default_factory=set)
2696:     priority: int = 0
2697:
2698:     def check_activation(self, method_characteristics: Dict) -> bool:
2699:         """
2700:             Check if this layer should be active given method characteristics.
2701:
2702:             Args:
2703:                 method_characteristics: Dict describing method properties
2704:
2705:             Returns:
2706:                 True if layer should be active
2707: """
2708:     return any(trigger(method_characteristics) for trigger in self.triggers)
2709:
2710:
2711: class LayerCoexistenceModel:
2712: """
2713:     Formal model of layer interactions and dependencies.
2714:
2715:     This encodes the complete "Layer Coexistence and Influence" system,
2716:     including:
2717:     - How to determine L(M) from method properties
2718:     - How layers influence each other
2719:     - Valid coexistence patterns
2720:     - Composition rules for multi-layer fusion
2721:
2722:     Design Principle: All layer interactions are explicit, not implicit.
2723:     No hidden dependencies or undocumented couplings permitted.
2724: """
2725:
2726:     def __init__(self):
2727:         self.influences: List[LayerInfluence] = []
2728:         self.activation_rules: Dict[Layer, LayerActivationRule] = {}
2729:         self.compatibility_matrix: Dict[Tuple[Layer, Layer], float] = {}
2730:
2731:         # Initialize canonical layer relationships
2732:         self._initialize_canonical_relationships()
2733:
2734:     def _initialize_canonical_relationships(self):
2735:         """
2736:             Define canonical layer relationships based on theoretical model.
2737:
2738:             Canonical Relationships:
2739:
2740:             1. @q \206\222 @d (WEIGHTING): Question-level evidence weights dimension scores
2741:                 - High certainty at @q increases weight of @d
2742:                 - Functional form: w_d' = w_d * (1 + f_A * certainty_q)
2743:
2744:             2. @d \206\222 @p (ACTIVATION): Dimensions determine relevant policy areas
```

```

2745:     - If dimension scores exist, policy coherence becomes relevant
2746:     - Functional form: active(@p)  $\geq$  |scored_dimensions|  $\geq$  threshold
2747:
2748:     3. @p  $\rightarrow$  @C (CONSTRAINT): Policy areas constrain ensemble methods
2749:         - Policy structure limits valid ensemble combinations
2750:         - Functional form: valid_ensembles  $\geq$  compatible_with_policy_structure
2751:
2752:     4. {@q, @d, @p}  $\rightarrow$  @m (TRANSFORMATION): Base layers feed meta-aggregation
2753:         - Meta layer synthesizes across evidence levels
2754:         - Functional form:  $x_m = g(x_q, x_d, x_p)$  where g is aggregation
2755:
2756:     5. @C  $\rightarrow$  @m (WEIGHTING): Congruence modulates meta-layer confidence
2757:         - Ensemble agreement increases meta-layer weight
2758:         - Functional form:  $w_m' = w_m \cdot \text{congruence\_score}$ 
2759: """
2760:
2761:     # @q  $\rightarrow$  @d influence
2762:     self.add_influence(LayerInfluence(
2763:         source_layer=Layer.QUESTION,
2764:         target_layer=Layer.DIMENSION,
2765:         influence_type=LayerInfluenceType.WEIGHTING,
2766:         strength=0.5,
2767:         functional_form="w_d' = w_d * (1 + 0.5 * certainty_q)",
2768:         conditions={'requires': 'question_certainty_available'}
2769:     ))
2770:
2771:     # @d  $\rightarrow$  @p influence
2772:     self.add_influence(LayerInfluence(
2773:         source_layer=Layer.DIMENSION,
2774:         target_layer=Layer.POLICY_AREA,
2775:         influence_type=LayerInfluenceType.ACTIVATION,
2776:         strength=1.0,
2777:         functional_form="active(@p)  $\geq$  |scored_dimensions|  $\geq$  3",
2778:         conditions={'threshold': 3}
2779:     ))
2780:
2781:     # @p  $\rightarrow$  @C influence
2782:     self.add_influence(LayerInfluence(
2783:         source_layer=Layer.POLICY_AREA,
2784:         target_layer=Layer.CONGRUENCE,
2785:         influence_type=LayerInfluenceType.CONSTRAINT,
2786:         strength=0.7,
2787:         functional_form="valid_ensembles  $\geq$  policy_compatible_ensembles",
2788:         conditions={'requires': 'policy_structure_defined'}
2789:     ))
2790:
2791:     # Base layers  $\rightarrow$  @m influence
2792:     for base_layer in [Layer.QUESTION, Layer.DIMENSION, Layer.POLICY_AREA]:
2793:         self.add_influence(LayerInfluence(
2794:             source_layer=base_layer,
2795:             target_layer=Layer.META,
2796:             influence_type=LayerInfluenceType.TRANSFORMATION,
2797:             strength=0.33,
2798:             functional_form="x_m = weighted_mean(x_q, x_d, x_p)",
2799:             conditions={'aggregation_type': 'weighted_mean'}
2800:         ))

```

```

2801:
2802:     # @C â\206\222 @m influence
2803:     self.add_influence(LayerInfluence(
2804:         source_layer=Layer.CONGRUENCE,
2805:         target_layer=Layer.META,
2806:         influence_type=LayerInfluenceType.WEIGHTING,
2807:         strength=0.6,
2808:         functional_form="w_m' = w_m * congruence_score",
2809:         conditions={'requires': 'ensemble_agreement'}
2810:     ))
2811:
2812:     # Initialize compatibility matrix (all pairs compatible by default)
2813:     for layer1 in Layer:
2814:         for layer2 in Layer:
2815:             # Diagonal: perfect self-compatibility
2816:             if layer1 == layer2:
2817:                 self.compatibility_matrix[(layer1, layer2)] = 1.0
2818:             # Off-diagonal: initialize to compatible
2819:             else:
2820:                 self.compatibility_matrix[(layer1, layer2)] = 0.8
2821:
2822:     # Adjust specific incompatibilities if any
2823:     # (Currently all layers are mutually compatible)
2824:
2825:     def add_influence(self, influence: LayerInfluence):
2826:         """Register a layer influence relationship."""
2827:         self.influences.append(influence)
2828:
2829:     def add_activation_rule(self, rule: LayerActivationRule):
2830:         """Register a layer activation rule."""
2831:         self.activation_rules[rule.layer] = rule
2832:
2833:     def determine_active_layers(
2834:         self,
2835:         method_characteristics: Dict
2836:     ) -> Set[Layer]:
2837:         """
2838:             Determine L(M) endogenously from method characteristics.
2839:
2840:             This is the key function that derives active layers from method
2841:             properties rather than requiring manual specification.
2842:
2843:             Args:
2844:                 method_characteristics: Dict with keys like:
2845:                     - 'operates_on_questions': bool
2846:                     - 'aggregates_dimensions': bool
2847:                     - 'addresses_policy_areas': bool
2848:                     - 'uses_ensemble': bool
2849:                     - 'performs_meta_aggregation': bool
2850:                     - 'question_count': int
2851:                     - 'dimension_count': int
2852:                     - 'policy_area_count': int
2853:
2854:             Returns:
2855:                 Set of active layers for this method
2856:         """

```

```
2857:         active = set()
2858:
2859:         # Sort rules by priority
2860:         sorted_rules = sorted(
2861:             self.activation_rules.items(),
2862:             key=lambda x: x[1].priority,
2863:             reverse=True
2864:         )
2865:
2866:         # Check each rule
2867:         for layer, rule in sorted_rules:
2868:             # Check prerequisites
2869:             if not rule.prerequisites.issubset(active):
2870:                 continue
2871:
2872:             # Check triggers
2873:             if rule.check_activation(method_characteristics):
2874:                 active.add(layer)
2875:
2876:     return active
2877:
2878:     def get_layer_influences(
2879:         self,
2880:         target_layer: Layer,
2881:         active_layers: Set[Layer]
2882:     ) -> List[LayerInfluence]:
2883:         """
2884:             Get all influences affecting a target layer from active layers.
2885:
2886:             Args:
2887:                 target_layer: Layer being influenced
2888:                 active_layers: Set of currently active layers
2889:
2890:             Returns:
2891:                 List of applicable LayerInfluence objects
2892:         """
2893:         return [
2894:             inf for inf in self.influences
2895:             if inf.target_layer == target_layer
2896:             and inf.source_layer in active_layers
2897:         ]
2898:
2899:     def compute_effective_weight(
2900:         self,
2901:         target_layer: Layer,
2902:         base_weight: float,
2903:         layer_scores: Dict[Layer, LayerScore],
2904:         active_layers: Set[Layer]
2905:     ) -> float:
2906:         """
2907:             Compute effective weight for a layer after applying influences.
2908:
2909:             Args:
2910:                 target_layer: Layer whose weight is being computed
2911:                 base_weight: Initial weight
2912:                 layer_scores: Scores for all active layers
```

```
2913:         active_layers: Set of active layers
2914:
2915:     Returns:
2916:         Effective weight after influence application
2917:
2918:     """
2919:     effective_weight = base_weight
2920:
2921:     # Get weighting influences
2922:     influences = [
2923:         inf for inf in self.get_layer_influences(target_layer, active_layers)
2924:         if inf.influence_type == LayerInfluenceType.WEIGHTING
2925:     ]
2926:
2927:     for influence in influences:
2928:         source_score = layer_scores.get(influence.source_layer)
2929:         if source_score:
2930:             # Apply influence (simplified model)
2931:             modifier = 1.0 + influence.strength * (source_score.value - 0.5)
2932:             effective_weight *= modifier
2933:
2934:     # Ensure weight stays in valid range
2935:     return max(0.0, min(1.0, effective_weight))
2936:
2937:     def check_compatibility(
2938:         self,
2939:         layer_set: Set[Layer]
2940:     ) -> Tuple[bool, float]:
2941:
2942:         """
2943:             Check if a set of layers is compatible for coexistence.
2944:
2945:         Args:
2946:             layer_set: Set of layers to check
2947:
2948:         Returns:
2949:             (is_compatible, compatibility_score)
2950:             where compatibility_score in [0, 1]
2951:
2952:         if len(layer_set) <= 1:
2953:             return True, 1.0
2954:
2955:         # Compute minimum pairwise compatibility
2956:         min_compatibility = 1.0
2957:         layer_list = list(layer_set)
2958:
2959:         for i, layer1 in enumerate(layer_list):
2960:             for layer2 in layer_list[i+1:]:
2961:                 compat = self.compatibility_matrix.get((layer1, layer2), 0.8)
2962:                 min_compatibility = min(min_compatibility, compat)
2963:
2964:         # Compatible if minimum exceeds threshold
2965:         is_compatible = min_compatibility >= 0.5
2966:         return is_compatible, min_compatibility
2967:
2968:     def export_model(self) -> Dict:
2969:         """Export model to JSON-serializable format."""
2970:         return {
```

```

2969:         'influences': [
2970:             {
2971:                 'source': inf.source_layer.value,
2972:                 'target': inf.target_layer.value,
2973:                 'type': inf.influence_type.value,
2974:                 'strength': inf.strength,
2975:                 'formula': inf.functional_form,
2976:                 'conditions': inf.conditions
2977:             }
2978:             for inf in self.influences
2979:         ],
2980:         'compatibility_matrix': {
2981:             f"{l1.value},{l2.value}": score
2982:             for (l1, l2), score in self.compatibility_matrix.items()
2983:         }
2984:     }
2985:
2986:
2987: def initialize_canonical_activation_rules() -> Dict[Layer, LayerActivationRule]:
2988: """
2989:     Initialize canonical activation rules for all layers.
2990:
2991:     These rules encode when each layer becomes relevant based on
2992:     method characteristics.
2993:
2994:     Returns:
2995:         Dict mapping Layer to LayerActivationRule
2996: """
2997: rules = {}
2998:
2999: # @q activation: Method operates on individual questions
3000: rules[Layer.QUESTION] = LayerActivationRule(
3001:     layer=Layer.QUESTION,
3002:     triggers=[
3003:         lambda mc: mc.get('operates_on_questions', False),
3004:         lambda mc: mc.get('question_count', 0) > 0,
3005:     ],
3006:     priority=100  # Highest priority - foundational layer
3007: )
3008:
3009: # @d activation: Method aggregates across dimensions
3010: rules[Layer.DIMENSION] = LayerActivationRule(
3011:     layer=Layer.DIMENSION,
3012:     triggers=[
3013:         lambda mc: mc.get('aggregates_dimensions', False),
3014:         lambda mc: mc.get('dimension_count', 0) > 0,
3015:     ],
3016:     prerequisites={Layer.QUESTION},  # Requires question layer
3017:     priority=90
3018: )
3019:
3020: # @p activation: Method addresses policy areas
3021: rules[Layer.POLICY_AREA] = LayerActivationRule(
3022:     layer=Layer.POLICY_AREA,
3023:     triggers=[
3024:         lambda mc: mc.get('addresses_policy_areas', False),

```

```
3025:         lambda mc: mc.get('policy_area_count', 0) > 0,
3026:         lambda mc: mc.get('dimension_count', 0) >= 3, # @d \206\222 @p influence
3027:     ],
3028:     prerequisites={Layer.DIMENSION}, # Requires dimension layer
3029:     priority=80
3030: )
3031:
3032: # @C activation: Method uses ensemble techniques
3033: rules[Layer.CONGRUENCE] = LayerActivationRule(
3034:     layer=Layer.CONGRUENCE,
3035:     triggers=[
3036:         lambda mc: mc.get('uses_ensemble', False),
3037:         lambda mc: mc.get('ensemble_method_count', 0) > 1,
3038:     ],
3039:     priority=70
3040: )
3041:
3042: # @m activation: Method performs cross-layer meta-aggregation
3043: rules[Layer.META] = LayerActivationRule(
3044:     layer=Layer.META,
3045:     triggers=[
3046:         lambda mc: mc.get('performs_meta_aggregation', False),
3047:         lambda mc: len(mc.get('active_base_layers', set())) >= 2,
3048:     ],
3049:     priority=60 # Lowest priority - synthesizes other layers
3050: )
3051:
3052: return rules
3053:
3054:
3055: # Global canonical model instance
3056: CANONICAL_LAYER_MODEL = LayerCoexistenceModel()
3057:
3058: # Register activation rules
3059: for layer, rule in initialize_canonical_activation_rules().items():
3060:     CANONICAL_LAYER_MODEL.add_activation_rule(rule)
3061:
3062:
3063:
3064: =====
3065: FILE: src/farfan_pipeline/core/calibration/meta_layer.py
3066: =====
3067:
3068: """
3069: Meta Layer (@m) - Full Implementation.
3070:
3071: Evaluates governance compliance using weighted formula:
3072: x_@m = 0.5@m_transp + 0.4@m_gov + 0.1@m_cost
3073: """
3074: import logging
3075: from typing import Optional
3076: from farfan_pipeline.core.calibration.config import MetaLayerConfig
3077:
3078: logger = logging.getLogger(__name__)
3079:
3080:
```

```
3081: class MetaLayerEvaluator:
3082:     """
3083:         Evaluates governance and meta-properties of methods.
3084:
3085:         Attributes:
3086:             config: MetaLayerConfig with weights and thresholds
3087:         """
3088:
3089:     def __init__(self, config: MetaLayerConfig):
3090:         """
3091:             Initialize evaluator with meta layer config.
3092:
3093:             Args:
3094:                 config: MetaLayerConfig instance
3095:             """
3096:         self.config = config
3097:         logger.info(
3098:             "meta_evaluator_initialized",
3099:             extra={
3100:                 "w_transparency": config.w_transparency,
3101:                 "w_governance": config.w_governance,
3102:                 "w_cost": config.w_cost
3103:             }
3104:         )
3105:
3106:     def evaluate(
3107:         self,
3108:         method_id: str,
3109:         method_version: str,
3110:         config_hash: str,
3111:         formula_exported: bool = False,
3112:         full_trace: bool = False,
3113:         logs_conform: bool = False,
3114:         signature_valid: bool = False,
3115:         execution_time_s: Optional[float] = None
3116:     ) -> float:
3117:         """
3118:             Compute the weighted score  $x_{@m} = w_{transparency} \cdot m_{transp} + w_{governance} \cdot m_{gov} + w_{cost} \cdot m_{cost}$ ,
3119:             where 'w_transparency', 'w_governance', and 'w_cost' come from the provided 'config'.
3120:
3121:             Args:
3122:                 method_id: Method identifier
3123:                 method_version: Method version string
3124:                 config_hash: Configuration hash
3125:                 formula_exported: Has formula been documented?
3126:                 full_trace: Is full execution trace available?
3127:                 logs_conform: Do logs conform to standard?
3128:                 signature_valid: Is cryptographic signature valid?
3129:                 execution_time_s: Runtime in seconds
3130:
3131:             Returns:
3132:                  $x_{@m} \in [0.0, 1.0]$ 
3133:             """
3134:         logger.info(
3135:             "meta_evaluation_start",
3136:             extra={
```

```
3137:             "method": method_id,
3138:             "version": method_version
3139:         }
3140:     )
3141:
3142:     # Component 1: Transparency (m_transp)
3143:     m_transp = self._compute_transparency(
3144:         formula_exported, full_trace, logs_conform
3145:     )
3146:     logger.debug("m_transp_computed", extra={"score": m_transp})
3147:
3148:     # Component 2: Governance (m_gov)
3149:     m_gov = self._compute_governance(
3150:         method_version, config_hash, signature_valid
3151:     )
3152:     logger.debug("m_gov_computed", extra={"score": m_gov})
3153:
3154:     # Component 3: Cost (m_cost)
3155:     m_cost = self._compute_cost(execution_time_s)
3156:     logger.debug("m_cost_computed", extra={"score": m_cost})
3157:
3158:     # Weighted sum
3159:     x_m = (
3160:         self.config.w_transparency * m_transp +
3161:         self.config.w_governance * m_gov +
3162:         self.config.w_cost * m_cost
3163:     )
3164:
3165:     logger.info(
3166:         "meta_computed",
3167:         extra={
3168:             "x_m": x_m,
3169:             "m_transp": m_transp,
3170:             "m_gov": m_gov,
3171:             "m_cost": m_cost,
3172:             "method": method_id
3173:         }
3174:     )
3175:
3176:     return x_m
3177:
3178: def _compute_transparency(
3179:     self,
3180:     formula: bool,
3181:     trace: bool,
3182:     logs: bool
3183: ) -> float:
3184:     """
3185:     Compute m_transp based on observability.
3186:
3187:     Scoring:
3188:         1.0: All 3 conditions met
3189:         0.7: 2/3 conditions met
3190:         0.4: 1/3 conditions met
3191:         0.0: 0/3 conditions met
3192:
```

```
3193:     Returns:
3194:         m_transp \210\210 {0.0, 0.4, 0.7, 1.0}
3195:     """
3196:     count = sum([formula, trace, logs])
3197:
3198:     if count == 3:
3199:         return 1.0
3200:     elif count == 2:
3201:         return 0.7
3202:     elif count == 1:
3203:         return 0.4
3204:     else:
3205:         return 0.0
3206:
3207:     def _compute_governance(
3208:         self,
3209:         version: str,
3210:         config_hash: str,
3211:         signature: bool
3212:     ) -> float:
3213:         """
3214:             Compute m_gov based on governance compliance.
3215:
3216:             Scoring:
3217:                 1.0: All 3 conditions met
3218:                 0.66: 2/3 conditions met
3219:                 0.33: 1/3 conditions met
3220:                 0.0: 0/3 conditions met
3221:
3222:             Returns:
3223:                 m_gov \210\210 {0.0, 0.33, 0.66, 1.0}
3224:             """
3225:             # Check version
3226:             has_version = bool(version and version != "unknown" and version != "1.0")
3227:
3228:             # Check config hash
3229:             has_hash = bool(config_hash and len(config_hash) > 0)
3230:
3231:             # Count conditions
3232:             count = sum([has_version, has_hash, signature])
3233:
3234:             if count == 3:
3235:                 return 1.0
3236:             elif count == 2:
3237:                 return 0.66
3238:             elif count == 1:
3239:                 return 0.33
3240:             else:
3241:                 return 0.0
3242:
3243:     def _compute_cost(self, execution_time_s: Optional[float] = None) -> float:
3244:         """
3245:             Compute m_cost based on runtime.
3246:
3247:             Scoring:
3248:                 1.0: < threshold_fast (e.g., <1s)
```

```
3249:         0.8: < threshold_acceptable (e.g., <5s)
3250:         0.5: >= threshold_acceptable
3251:         0.0: timeout/OOM (not provided)
3252:
3253:     Returns:
3254:         m_cost â\210\210 {0.0, 0.5, 0.8, 1.0}
3255:     """
3256:     if execution_time_s is None:
3257:         logger.warning("meta_timing_not_available")
3258:         return 0.5 # Default: acceptable
3259:
3260:     if execution_time_s < 0:
3261:         logger.error("meta_negative_time", extra={"time": execution_time_s})
3262:         return 0.0
3263:
3264:     if execution_time_s < self.config.threshold_fast:
3265:         return 1.0 # Fast
3266:     elif execution_time_s < self.config.threshold_acceptable:
3267:         return 0.8 # Acceptable
3268:     else:
3269:         logger.warning(
3270:             "meta_slow_execution",
3271:             extra={
3272:                 "runtime": execution_time_s,
3273:                 "threshold": self.config.threshold_acceptable
3274:             }
3275:         )
3276:         return 0.5 # Slow but usable
3277:
3278:
3279:
3280: =====
3281: FILE: src/farfan_pipeline/core/calibration/orchestrator.py
3282: =====
3283:
3284: """Calibration Orchestrator - Mandatory Single Path Enforcement.
3285:
3286: This module implements the ONLY allowed path for calibration scoring.
3287: Any code that bypasses this orchestrator MUST be rejected.
3288:
3289: Architecture:
3290: - Singleton pattern enforces single entry point
3291: - Loads intrinsic calibration (@b scores) from JSON
3292: - Reads layer requirements from canonical inventory
3293: - Computes runtime layers dynamically (@chain, @q, @d, @p, @C, @u, @m)
3294: - Aggregates via choquet_integral or weighted_sum
3295: - Applies threshold: â\211¥0.7 PASS, <0.7 FAIL
3296:
3297: CRITICAL: This is the ONLY calibration path. All other calibration logic must be removed.
3298: """
3299:
3300: from __future__ import annotations
3301:
3302: import json
3303: import logging
3304: from pathlib import Path
```

```
3305: from typing import TYPE_CHECKING
3306:
3307: from farfan_pipeline.core.orchestrator.calibration_types import (
3308:     IntrinsicScores,
3309:     LayerRequirements,
3310:     RuntimeLayers,
3311: )
3312:
3313: if TYPE_CHECKING:
3314:     from farfan_pipeline.core.orchestrator.calibration_context import CalibrationContext
3315:
3316: logger = logging.getLogger(__name__)
3317:
3318: CALIBRATION_THRESHOLD = 0.7
3319:
3320:
3321: class MissingIntrinsicCalibrationError(Exception):
3322:     """Raised when intrinsic calibration (@b scores) are missing for a method."""
3323:
3324:     def __init__(self, method_id: str) -> None:
3325:         self.method_id = method_id
3326:         super().__init__(
3327:             f"Missing intrinsic calibration for method '{method_id}'. "
3328:             f"All methods must have @b scores in intrinsic_calibration.json"
3329:         )
3330:
3331:
3332: class InsufficientContextError(Exception):
3333:     """Raised when context is required but not provided."""
3334:
3335:     def __init__(self, method_id: str) -> None:
3336:         self.method_id = method_id
3337:         super().__init__(
3338:             f"Insufficient context for method '{method_id}'. "
3339:             f"CalibrationContext is required for runtime layer evaluation."
3340:         )
3341:
3342:
3343: class MethodBelowThresholdError(Exception):
3344:     """Raised when a method's calibration score falls below threshold."""
3345:
3346:     def __init__(self, method_id: str, score: float, threshold: float) -> None:
3347:         self.method_id = method_id
3348:         self.score = score
3349:         self.threshold = threshold
3350:         super().__init__(
3351:             f"Method '{method_id}' calibration score {score:.3f} is below "
3352:             f"threshold {threshold:.3f}. Method cannot be executed."
3353:         )
3354:
3355:
3356: class CalibrationOrchestrator:
3357:     """Singleton orchestrator for all calibration operations.
3358:
3359:     This is the MANDATORY and ONLY path for calibration scoring.
3360:     """
3361:
```

```
3361:
3362:     _instance: CalibrationOrchestrator | None = None
3363:     _initialized: bool = False
3364:
3365:     def __new__(cls) -> CalibrationOrchestrator:
3366:         if cls._instance is None:
3367:             cls._instance = super().__new__(cls)
3368:         return cls._instance
3369:
3370:     def __init__(self) -> None:
3371:         if self._initialized:
3372:             return
3373:
3374:         self._intrinsic_scores: dict[str, IntrinsicScores] = {}
3375:         self._layer_requirements: dict[str, LayerRequirements] = {}
3376:         self._runtime_layer_config: dict = {}
3377:         self._load_intrinsic_calibration()
3378:         self._load_layer_requirements()
3379:         self._load_runtime_layer_config()
3380:
3381:         CalibrationOrchestrator._initialized = True
3382:         logger.info("CalibrationOrchestrator initialized (singleton)")
3383:
3384:     def _load_intrinsic_calibration(self) -> None:
3385:         """Load intrinsic calibration using IntrinsicCalibrationLoader."""
3386:         from farfan_pipeline.core.calibration.intrinsic_calibration_loader import get_intrinsic_calibration_loader
3387:
3388:         loader = get_intrinsic_calibration_loader()
3389:
3390:         # Get all methods from intrinsic calibration
3391:         # Load the JSON directly to get all method IDs
3392:         import json
3393:         from pathlib import Path
3394:
3395:         config_path = Path("config/intrinsic_calibration.json")
3396:         if not config_path.exists():
3397:             logger.warning(f"Intrinsic calibration file not found: {config_path}")
3398:             return
3399:
3400:         try:
3401:             with open(config_path, encoding='utf-8') as f:
3402:                 data = json.load(f)
3403:
3404:                 for method_id in data.keys():
3405:                     if method_id == "_metadata":
3406:                         continue
3407:
3408:                         calibration = loader.get_calibration(method_id)
3409:                         if calibration is not None:
3410:                             self._intrinsic_scores[method_id] = IntrinsicScores(
3411:                                 b_theory=calibration.b_theory,
3412:                                 b_impl=calibration.b_impl,
3413:                                 b_deploy=calibration.b_deploy
3414:                             )
3415:
3416:             logger.info(f"Loaded intrinsic scores for {len(self._intrinsic_scores)} methods")
```

```
3417:  
3418:         except Exception as e:  
3419:             logger.error(f"Failed to load intrinsic calibration: {e}")  
3420:             raise  
3421:  
3422:     def _load_layer_requirements(self) -> None:  
3423:         """Load layer requirements from layer_assignment module."""  
3424:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS, CHOQUET_WEIGHTS  
3425:  
3426:         # For each method in intrinsic calibration, assign layers  
3427:         for method_id in self._intrinsic_scores.keys():  
3428:             # Determine role from method_id or from intrinsic calibration  
3429:             role = self._determine_role(method_id)  
3430:  
3431:             if role in LAYER_REQUIREMENTS:  
3432:                 required_layers = LAYER_REQUIREMENTS[role]  
3433:  
3434:                 # Build weights dict  
3435:                 weights = {}  
3436:                 for layer in required_layers:  
3437:                     if layer in CHOQUET_WEIGHTS:  
3438:                         weights[layer] = CHOQUET_WEIGHTS[layer]  
3439:  
3440:                 # Normalize weights  
3441:                 total = sum(weights.values())  
3442:                 if total > 0:  
3443:                     weights = {k: v/total for k, v in weights.items()}  
3444:  
3445:                 self._layer_requirements[method_id] = LayerRequirements(  
3446:                     required_layers=required_layers,  
3447:                     weights=weights,  
3448:                     aggregation_method='weighted_sum'  
3449:                 )  
3450:  
3451:             logger.info(f"Loaded layer requirements for {len(self._layer_requirements)} methods")  
3452:  
3453:     def _determine_role(self, method_id: str) -> str:  
3454:         """Determine role from method_id or intrinsic calibration."""  
3455:         # If it's an executor (D1Q1, etc), it's 'executor'  
3456:         if 'D' in method_id and 'Q' in method_id:  
3457:             return 'executor'  
3458:  
3459:         # Try to get from intrinsic calibration  
3460:         from farfan_pipeline.core.calibration.intrinsic_calibration_loader import get_intrinsic_calibration_loader  
3461:         loader = get_intrinsic_calibration_loader()  
3462:         calibration = loader.get_calibration(method_id)  
3463:  
3464:         from farfan_pipeline.core.calibration.layer_assignment import LAYER_REQUIREMENTS  
3465:         if calibration and hasattr(calibration, 'layer') and calibration.layer in LAYER_REQUIREMENTS:  
3466:             return calibration.layer  
3467:  
3468:         # Default to analyzer (safest - uses all layers)  
3469:         return 'analyzer'  
3470:  
3471:     def _load_runtime_layer_config(self) -> None:  
3472:         """Load runtime layer configuration from system/config/calibration/runtime_layers.json."""
```

```
3473:     config_path = Path("system/config/calibration/runtime_layers.json")
3474:     if not config_path.exists():
3475:         logger.warning(f"Runtime layer config not found: {config_path}, using defaults")
3476:         return
3477:
3478:     try:
3479:         with open(config_path, encoding='utf-8') as f:
3480:             self._runtime_layer_config = json.load(f)
3481:             logger.info("Loaded runtime layer configuration")
3482:         except Exception as e:
3483:             logger.error(f"Failed to load runtime layer config: {e}")
3484:             raise
3485:
3486:     def evaluate_runtime_layers(
3487:         self,
3488:         method_id: str,
3489:         context: CalibrationContext | None
3490:     ) -> RuntimeLayers:
3491:         """Evaluate runtime layers dynamically based on context.
3492:
3493:         Args:
3494:             method_id: Method identifier
3495:             context: Calibration context (required for evaluation)
3496:
3497:         Returns:
3498:             RuntimeLayers with computed scores
3499:
3500:         Raises:
3501:             InsufficientContextError: If context is None
3502:         """
3503:         if context is None:
3504:             raise InsufficientContextError(method_id)
3505:
3506:         chain_score = self._compute_chain_score(context)
3507:         quality_score = self._compute_quality_score(context)
3508:         density_score = self._compute_density_score(context)
3509:         provenance_score = self._compute_provenance_score(context)
3510:         coverage_score = self._compute_coverage_score(context)
3511:         uncertainty_score = self._compute_uncertainty_score(context)
3512:         mechanism_score = self._compute_mechanism_score(context)
3513:
3514:         layers = RuntimeLayers(
3515:             chain=chain_score,
3516:             quality=quality_score,
3517:             density=density_score,
3518:             provenance=provenance_score,
3519:             coverage=coverage_score,
3520:             uncertainty=uncertainty_score,
3521:             mechanism=mechanism_score
3522:         )
3523:
3524:         logger.debug(
3525:             f"Runtime layers for {method_id}: {layers.to_dict()}"
3526:         )
3527:
3528:     return layers
```

```
3529:  
3530:     def _compute_chain_score(self, context: CalibrationContext) -> float:  
3531:         """Compute chain of evidence score (@chain)."""  
3532:         config = self._runtime_layer_config.get('layers', {}).get('chain', {})  
3533:         base_score = config.get('base_score', 0.65)  
3534:         dimension_factor = config.get('dimension_factor', 0.15)  
3535:         dimension_max = config.get('dimension_max', 10.0)  
3536:         position_bonus = config.get('position_bonus', 0.1)  
3537:         position_threshold = config.get('position_threshold', 0.5)  
3538:  
3539:         score = base_score  
3540:         if context.dimension > 0:  
3541:             score += dimension_factor * min(context.dimension / dimension_max, 1.0)  
3542:         if context.method_position < context.total_methods * position_threshold:  
3543:             score += position_bonus  
3544:         return min(score, 1.0)  
3545:  
3546:     def _compute_quality_score(self, context: CalibrationContext) -> float:  
3547:         """Compute data quality score (@q)."""  
3548:         config = self._runtime_layer_config.get('layers', {}).get('quality', {})  
3549:         base_score = config.get('base_score', 0.70)  
3550:         question_factor = config.get('question_factor', 0.08)  
3551:         question_max = config.get('question_max', 20.0)  
3552:  
3553:         score = base_score  
3554:         if context.question_num > 0:  
3555:             score += question_factor * min(context.question_num / question_max, 1.0)  
3556:         return min(score, 1.0)  
3557:  
3558:     def _compute_density_score(self, context: CalibrationContext) -> float:  
3559:         """Compute data density score (@d)."""  
3560:         config = self._runtime_layer_config.get('layers', {}).get('density', {})  
3561:         base_score = config.get('base_score', 0.68)  
3562:         position_factor = config.get('position_factor', 0.15)  
3563:         optimal_position = config.get('optimal_position', 0.5)  
3564:  
3565:         score = base_score  
3566:         if context.total_methods > 0:  
3567:             ratio = context.method_position / context.total_methods  
3568:             score += position_factor * (1.0 - abs(optimal_position - ratio))  
3569:         return min(score, 1.0)  
3570:  
3571:     def _compute_provenance_score(self, context: CalibrationContext) -> float:  
3572:         """Compute provenance traceability score (@p)."""  
3573:         config = self._runtime_layer_config.get('layers', {}).get('provenance', {})  
3574:         return config.get('base_score', 0.75)  
3575:  
3576:     def _compute_coverage_score(self, context: CalibrationContext) -> float:  
3577:         """Compute coverage completeness score (@C)."""  
3578:         config = self._runtime_layer_config.get('layers', {}).get('coverage', {})  
3579:         base_score = config.get('base_score', 0.72)  
3580:         bonus_dimensions = config.get('bonus_dimensions', [1, 2, 5, 10])  
3581:         dimension_bonus = config.get('dimension_bonus', 0.1)  
3582:  
3583:         score = base_score  
3584:         if context.dimension in bonus_dimensions:
```

```
3585:         score += dimension_bonus
3586:     return min(score, 1.0)
3587:
3588:     def _compute_uncertainty_score(self, context: CalibrationContext) -> float:
3589:         """Compute uncertainty quantification score (@u)."""
3590:         config = self._runtime_layer_config.get('layers', {}).get('uncertainty', {})
3591:         return config.get('base_score', 0.68)
3592:
3593:     def _compute_mechanism_score(self, context: CalibrationContext) -> float:
3594:         """Compute mechanistic explanation score (@m)."""
3595:         config = self._runtime_layer_config.get('layers', {}).get('mechanism', {})
3596:         base_score = config.get('base_score', 0.65)
3597:         dimension_threshold = config.get('dimension_threshold', 7)
3598:         dimension_bonus = config.get('dimension_bonus', 0.15)
3599:
3600:         score = base_score
3601:         if context.dimension >= dimension_threshold:
3602:             score += dimension_bonus
3603:         return min(score, 1.0)
3604:
3605:     def choquet_integral(
3606:         self,
3607:         layers: RuntimeLayers,
3608:         weights: dict[str, float]
3609:     ) -> float:
3610:         """Aggregate layers using Choquet integral for executors.
3611:
3612:             Simplified Choquet-like aggregation that weights higher-scoring layers more heavily.
3613:
3614:             Args:
3615:                 layers: Runtime layer scores
3616:                 weights: Layer weights from requirements
3617:
3618:             Returns:
3619:                 Aggregated score (0.0-1.0)
3620: """
3621:         layer_dict = layers.to_dict()
3622:
3623:         total_weight = sum(weights.values())
3624:         if total_weight == 0:
3625:             return 0.0
3626:
3627:         weighted_total = sum(
3628:             layer_dict.get(name, 0.0) * weight
3629:             for name, weight in weights.items()
3630:         )
3631:
3632:         return min(max(weighted_total / total_weight, 0.0), 1.0)
3633:
3634:     def weighted_sum(
3635:         self,
3636:         layers: RuntimeLayers,
3637:         weights: dict[str, float]
3638:     ) -> float:
3639:         """Aggregate layers using weighted sum for non-executors.
3640:
```

```
3641:     Args:
3642:         layers: Runtime layer scores
3643:         weights: Layer weights from requirements
3644:
3645:     Returns:
3646:         Aggregated score (0.0-1.0)
3647:         """
3648:         layer_dict = layers.to_dict()
3649:         total_weight = sum(weights.values())
3650:
3651:         if total_weight == 0:
3652:             return 0.0
3653:
3654:         weighted_total = sum(
3655:             layer_dict.get(name, 0.0) * weight
3656:             for name, weight in weights.items()
3657:         )
3658:
3659:         return min(max(weighted_total / total_weight, 0.0), 1.0)
3660:
3661:     def calibrate_method(
3662:         self,
3663:         method_id: str,
3664:         context: CalibrationContext | None = None,
3665:         is_executor: bool = False
3666:     ) -> float:
3667:         """MANDATORY CALIBRATION PATH - This is the ONLY allowed entry point.
3668:
3669:         Args:
3670:             method_id: Method identifier
3671:             context: Calibration context (required for runtime evaluation)
3672:             is_executor: Whether this is an executor method (uses Choquet)
3673:
3674:         Returns:
3675:             Final calibration score (0.0-1.0)
3676:
3677:         Raises:
3678:             MissingIntrinsicCalibrationError: If @b scores missing
3679:             InsufficientContextError: If context is None
3680:             MethodBelowThresholdError: If score < threshold
3681:             """
3682:         if method_id not in self._intrinsic_scores:
3683:             raise MissingIntrinsicCalibrationError(method_id)
3684:
3685:         intrinsic = self._intrinsic_scores[method_id]
3686:         intrinsic_score = intrinsic.average()
3687:
3688:         logger.debug(
3689:             f"Intrinsic score for {method_id}: {intrinsic_score:.3f} "
3690:             f"(theory={intrinsic.b_theory:.3f}, impl={intrinsic.b_impl:.3f}, "
3691:             f"deploy={intrinsic.b_deploy:.3f})"
3692:         )
3693:
3694:         runtime_layers = self.evaluate_runtime_layers(method_id, context)
3695:
3696:         requirements = self._layer_requirements.get(method_id)
```

```
3697:         if requirements is None:
3698:             logger.warning(
3699:                 f"No layer requirements for {method_id}, using default weights"
3700:             )
3701:             requirements = LayerRequirements(
3702:                 required_layers=['quality', 'provenance'],
3703:                 weights={'quality': 0.5, 'provenance': 0.5},
3704:                 aggregation_method='weighted_sum'
3705:             )
3706:
3707:         if is_executor or requirements.aggregation_method == 'choquet_integral':
3708:             runtime_score = self.choquet_integral(runtime_layers, requirements.weights)
3709:         else:
3710:             runtime_score = self.weighted_sum(runtime_layers, requirements.weights)
3711:
3712:         final_score = (intrinsic_score + runtime_score) / 2.0
3713:
3714:         logger.info(
3715:             f"Calibration for {method_id}: intrinsic={intrinsic_score:.3f}, "
3716:             f"runtime={runtime_score:.3f}, final={final_score:.3f}"
3717:         )
3718:
3719:         if final_score < CALIBRATION_THRESHOLD:
3720:             raise MethodBelowThresholdError(method_id, final_score, CALIBRATION_THRESHOLD)
3721:
3722:     return final_score
3723:
3724:     @classmethod
3725:     def get_instance(cls) -> CalibrationOrchestrator:
3726:         """Get singleton instance."""
3727:         if cls._instance is None:
3728:             cls._instance = CalibrationOrchestrator()
3729:         return cls._instance
3730:
3731:     @classmethod
3732:     def reset_instance(cls) -> None:
3733:         """Reset singleton instance (for testing only)."""
3734:         cls._instance = None
3735:         cls._initialized = False
3736:
3737:
3738:     __all__ = [
3739:         'CalibrationOrchestrator',
3740:         'MissingIntrinsicCalibrationError',
3741:         'InsufficientContextError',
3742:         'MethodBelowThresholdError',
3743:         'CALIBRATION_THRESHOLD'
3744:     ]
3745:
3746:
3747:
3748: =====
3749: FILE: src/farfan_pipeline/core/calibration/parameter_loader.py
3750: =====
3751:
3752: """Legacy parameter loader - now wraps ParameterLoaderV2."""
```

```
3753:  
3754: from typing import Any  
3755:  
3756: from farfan_pipeline.core.parameters import ParameterLoaderV2  
3757:  
3758:  
3759: class ParameterLoader:  
3760:     """Stub parameter loader for backward compatibility.  
3761:  
3762:     Legacy wrapper around ParameterLoaderV2 for backward compatibility.  
3763:     DEPRECATED: Use ParameterLoaderV2.get(method_id, param_name) directly.  
3764:     """  
3765:  
3766:     def __init__(self) -> None:  
3767:         pass  
3768:  
3769:     def load(self) -> None:  
3770:         """No-op: ParameterLoaderV2 auto-loads."""  
3771:         pass  
3772:  
3773:     def get(  
3774:         self, method_id: str, default: dict[str, Any] | None = None  
3775:     ) -> dict[str, Any]:  
3776:         """  
3777:             Gets the parameters for a given method_id.  
3778:             Delegates to ParameterLoaderV2.  
3779:         """  
3780:         if default is None:  
3781:             default = {}  
3782:  
3783:         params = ParameterLoaderV2.get_all(method_id)  
3784:         return params if params else default  
3785:  
3786:  
3787: _parameter_loader = ParameterLoader()  
3788:  
3789:  
3790: def get_parameter_loader() -> ParameterLoader:  
3791:     """Get the global parameter loader instance."""  
3792:     return _parameter_loader  
3793:  
3794:  
3795:  
3796: =====  
3797: FILE: src/farfan_pipeline/core/calibration/pdt_structure.py  
3798: =====  
3799:  
3800: """  
3801: PDT (Plan de Desarrollo Territorial) structure definition.  
3802:  
3803: This module defines the data structure that represents a parsed PDT.  
3804: The PDT is the INPUT to the Unit Layer evaluation.  
3805:  
3806: The structure is populated by a separate PDT parser (not shown here),  
3807: which extracts:  
3808: - Text content and tokens
```

```
3809: - Block structure (Diagn stico, Estrat gica, PPI, Seguimiento)
3810: - Section analysis (keywords, numbers, sources)
3811: - Indicator matrix (if present)
3812: - PPI matrix (if present)
3813: """
3814: from dataclasses import dataclass, field
3815: from typing import Any
3816:
3817:
3818: @dataclass
3819: class PDTStructure:
3820: """
3821:     Extracted structure of a PDT.
3822:
3823:     This is populated by the PDT parser and consumed by UnitLayerEvaluator.
3824:
3825:     Attributes:
3826:         full_text: Complete text of the PDT
3827:         total_tokens: Total word/token count
3828:         blocks_found: Detected structural blocks
3829:         headers: List of headers with numbering validation
3830:         block_sequence: Actual order of blocks (for checking sequence)
3831:         sections_found: Analysis of mandatory sections
3832:         indicator_matrix_present: Whether indicator table was found
3833:         indicator_rows: Parsed indicator table rows
3834:         ppi_matrix_present: Whether PPI table was found
3835:         ppi_rows: Parsed PPI table rows
3836: """
3837: # Raw content
3838: full_text: str
3839: total_tokens: int
3840:
3841: # Block detection (for S - Structural compliance)
3842: blocks_found: dict[str, dict[str, Any]] = field(default_factory=dict)
3843: # Example:
3844: #     "Diagn stico": {"text": "...", "tokens": 1500, "numbers_count": 25},
3845: #     "Parte Estrat gica": {"text": "...", "tokens": 1200, "numbers_count": 15},
3846: # }
3847:
3848: headers: list[dict[str, Any]] = field(default_factory=list)
3849: # Example:
3850: #     {"level": 1, "text": "1. DIAGN STICO", "valid_numbering": True},
3851: #     {"level": 2, "text": "1.1 Contexto", "valid_numbering": True},
3852: # ]
3853:
3854: block_sequence: list[str] = field(default_factory=list)
3855: # Example: ["Diagn stico", "Parte Estrat gica", "PPI", "Seguimiento"]
3856:
3857: # Section analysis (for M - Mandatory sections)
3858: sections_found: dict[str, dict[str, Any]] = field(default_factory=dict)
3859: # Example:
3860: #     "Diagn stico": {
3861: #         "present": True,
3862: #         "token_count": 1500,
3863: #         "keyword_matches": 5, # e.g., "brecha", "DANE", "l -nea base"
3864: #         "number_count": 25,
```

```
3865:     #         "sources_found": 3, # e.g., "DANE", "Medicina Legal"
3866:     #     }
3867:     # }
3868:
3869:     # Indicator matrix (for I - Indicator quality)
3870:     indicator_matrix_present: bool = False
3871:     indicator_rows: list[dict[str, Any]] = field(default_factory=list)
3872:     # Example: [
3873:     #     {
3874:     #         "Tipo": "PRODUCTO",
3875:     #         "LÃnea EstratÃgica": "Equidad de GÃnnero",
3876:     #         "Programa": "PrevenciÃ³n de VBG",
3877:     #         "LÃnea Base": "120 casos",
3878:     #         "AÃ±o LB": 2023,
3879:     #         "Meta Cuatrienio": "80 casos",
3880:     #         "Fuente": "ComisarÃa de Familia",
3881:     #         "Unidad Medida": "Casos reportados",
3882:     #         "CÃdigo MGA": "1234567"
3883:     #     }
3884:     # ]
3885:
3886:     # PPI matrix (for P - PPI completeness)
3887:     ppi_matrix_present: bool = False
3888:     ppi_rows: list[dict[str, Any]] = field(default_factory=list)
3889:     # Example: [
3890:     #     {
3891:     #         "LÃnea EstratÃgica": "Equidad de GÃnnero",
3892:     #         "Programa": "PrevenciÃ³n de VBG",
3893:     #         "Costo Total": 500000000,
3894:     #         "2024": 100000000,
3895:     #         "2025": 150000000,
3896:     #         "2026": 150000000,
3897:     #         "2027": 100000000,
3898:     #         "SGP": 300000000,
3899:     #         "SGR": 0,
3900:     #         "Propios": 200000000,
3901:     #         "Otras": 0
3902:     #     }
3903:     # ]
3904:
3905:
3906:
3907: =====
3908: FILE: src/farfán_pipeline/core/calibration/rigorous_calibration_triage.py
3909: =====
3910:
3911: #!/usr/bin/env python3
3912: """
3913: Rigorous Intrinsic Calibration Triage - Method by Method Analysis
3914:
3915: Per tesislizayjuan-debug requirements (comments 3512949686, 3513311176):
3916: - Apply decision automaton to EVERY method in canonical_method_catalog.json
3917: - Use machine-readable rubric from config/intrinsic_calibration_rubric.json
3918: - Produce traceable, reproducible evidence for all scores
3919:
3920: Pass 1: Determine if method requires calibration (3-question gate per rubric)
```

```
3921: Pass 2: Compute evidence-based intrinsic scores using explicit rubric rules
3922: Pass 3: Populate intrinsic_calibration.json with reproducible evidence
3923:
3924: NO UNIFORM DEFAULTS. Each method analyzed individually.
3925: ALL SCORES TRACEABLE. Evidence shows exact computation path.
3926: """
3927:
3928: import json
3929: import sys
3930: import ast
3931: import re
3932: from pathlib import Path
3933: from datetime import datetime, timezone
3934: from typing import Dict, Any, Tuple, Optional, List
3935:
3936:
3937: def load_json(path: Path) -> dict:
3938:     """Load JSON file"""
3939:     with open(path, 'r') as f:
3940:         return json.load(f)
3941:
3942:
3943: def save_json(path: Path, data: dict) -> None:
3944:     """Save JSON file with formatting"""
3945:     with open(path, 'w') as f:
3946:         json.dump(data, f, indent=2, ensure_ascii=False)
3947:         f.write('\n')
3948:
3949:
3950: def triage_pass1_requires_calibration(method_info: Dict[str, Any], rubric: Dict[str, Any]) -> Tuple[bool, str, Dict[str, Any]]:
3951: """
3952:     Pass 1: Does this method require intrinsic calibration?
3953:     Determine if a method requires calibration based on the 3-question rubric.
3954:
3955:     Returns: (requires_calibration, reason, evidence_dict)
3956: """
3957:     method_name = method_info.get('method_name', '')
3958:     docstring = method_info.get('docstring', '') or ''
3959:     layer = method_info.get('layer', 'unknown')
3960:     return_type = method_info.get('return_type', '')
3961:
3962:     # Load decision rules from rubric
3963:     triggers = rubric['calibration_triggers']
3964:     exclusion_rules = rubric['exclusion_criteria']
3965:
3966:     # Check explicit exclusion patterns first
3967:     exclusion_patterns = exclusion_rules['patterns']
3968:     for pattern_rule in exclusion_patterns:
3969:         if pattern_rule['pattern'] in method_name:
3970:             return False, pattern_rule['reason'], {
3971:                 "matched_exclusion_pattern": pattern_rule['pattern'],
3972:                 "exclusion_reason": pattern_rule['reason']
3973:             }
3974:
3975:     # Q1: Analytically active?
3976:     q1_config = triggers['questions'][q1_analytically_active']
```

```

3977:     primary_verbs = q1_config['indicators'].get('primary_analytical_verbs', [])
3978:     etl_verbs = q1_config['indicators'].get('generative_etl_verbs', [])
3979:     all_analytical_verbs = primary_verbs + etl_verbs
3980:
3981:     q1_matches_name = [verb for verb in all_analytical_verbs if verb in method_name.lower()]
3982:     q1_matches_doc = [verb for verb in all_analytical_verbs[:10] if verb in docstring.lower()]
3983:     q1_analytical = len(q1_matches_name) > 0 or len(q1_matches_doc) > 0
3984:
3985:     # Q2: Parametric?
3986:     q2_config = triggers['questions']['q2_parametric']
3987:     parametric_keywords = q2_config['indicators'].get('parametric_keywords', [])
3988:     parametric_verbs = q2_config['indicators'].get('parametric_verbs', [])
3989:     critical_layers = q2_config['indicators'].get('check_layer', [])
3990:
3991:     q2_matches_kw = [kw for kw in parametric_keywords if kw in docstring.lower()]
3992:     q2_matches_verb = [verb for verb in parametric_verbs if verb in method_name.lower()]
3993:     q2_parametric = len(q2_matches_kw) > 0 or len(q2_matches_verb) > 0 or layer in critical_layers
3994:
3995:     # Q3: Safety-critical?
3996:     q3_config = triggers['questions']['q3_safety_critical']
3997:     safety_verbs = q3_config['indicators'].get('safety_verbs', [])
3998:     safety_layers = q3_config['indicators'].get('critical_layers', [])
3999:     eval_types = q3_config['indicators'].get('evaluative_return_types', [])
4000:
4001:     q3_matches_verb = [verb for verb in safety_verbs if verb in method_name.lower()]
4002:     q3_safety_critical = (len(q3_matches_verb) > 0 or
4003:                           layer in safety_layers or
4004:                           return_type in eval_types)
4005:
4006:     if q3_config['indicators'].get('exclude_simple_getters', False) and method_name.startswith('get_'):
4007:         # Only exclude if it's NOT a safety verb (e.g. 'get_validation_status' might be critical, but 'get_name' is not)
4008:         if not q3_matches_verb:
4009:             q3_safety_critical = False
4010:
4011:     # Additional exclusion rules
4012:     is_private_utility = (method_name.startswith('_') and
4013:                           not q1_analytical and
4014:                           layer == 'utility')
4015:     is_pure_getter = (method_name.startswith('get_') and
4016:                           return_type in ['str', 'Path', 'bool'] and
4017:                           not q1_analytical and not q3_safety_critical)
4018:
4019:     # Build machine-readable evidence
4020:     triage_evidence = {
4021:         "q1_analytically_active": {
4022:             "result": q1_analytical,
4023:             "matched_verbs_in_name": q1_matches_name,
4024:             "matched_verbs_in_doc": q1_matches_doc
4025:         },
4026:         "q2_parametric": {
4027:             "result": q2_parametric,
4028:             "matched_keywords": q2_matches_kw,
4029:             "matched_verbs": q2_matches_verb,
4030:             "layer_is_critical": layer in critical_layers
4031:         },
4032:         "q3_safety_critical": {
4033:             "result": q3_safety_critical,
4034:             "matched_verbs": q3_matches_verb
4035:         }
4036:     }

```

```
4033:         "result": q3_safety_critical,
4034:         "matched_safety_verbs": q3_matches_verb,
4035:         "layer_is_critical": layer in safety_layers,
4036:         "return_type_is_evaluative": return_type in eval_types
4037:     },
4038:     "decision_rule": "requires_calibration = (q1 OR q2 OR q3) AND NOT excluded"
4039: }
4040:
4041: # Decision per rubric
4042: if is_private_utility:
4043:     return False, "Private utility function - non-analytical", triage_evidence
4044:
4045: if is_pure_getter:
4046:     return False, "Simple getter with no analytical logic", triage_evidence
4047:
4048: if q1_analytical or q2_parametric or q3_safety_critical:
4049:     reasons = []
4050:     if q1_analytical:
4051:         reasons.append("analytically active")
4052:     if q2_parametric:
4053:         reasons.append("encodes assumptions/knobs")
4054:     if q3_safety_critical:
4055:         reasons.append("safety-critical for evaluation")
4056:     return True, f"Requires calibration: {', '.join(reasons)}", triage_evidence
4057:
4058: return False, "Non-analytical utility function", triage_evidence
4059:
4060:
4061: def compute_b_theory(method_info: Dict[str, Any], repo_root: Path, rubric: Dict[str, Any]) -> Tuple[float, Dict]:
4062: """
4063: Compute b_theory: theoretical foundation quality
4064:
4065: Uses machine-readable rules from rubric config
4066:
4067: docstring = method_info.get('docstring', '') or ''
4068: method_name = method_info.get('method_name', '')
4069:
4070: # Load rubric rules
4071: b_theory_config = rubric['b_theory']
4072: weights = b_theory_config['weights']
4073: rules = b_theory_config['rules']
4074:
4075: # Component 1: Statistical grounding
4076: stat_rules = rules['grounded_in_valid_statistics']['scoring']
4077: stat_keywords = stat_rules['has_bayesian_or_statistical_model']['keywords']
4078: stat_matches = [kw for kw in stat_keywords if kw in docstring.lower()]
4079:
4080: if len(stat_matches) >= stat_rules['has_bayesian_or_statistical_model']['threshold']:
4081:     stat_score = stat_rules['has_bayesian_or_statistical_model']['score']
4082: elif len(stat_matches) >= stat_rules['has_some_statistical_grounding']['threshold']:
4083:     stat_score = stat_rules['has_some_statistical_grounding']['score']
4084: else:
4085:     stat_score = stat_rules['no_statistical_grounding']['score']
4086:
4087: # Component 2: Logical consistency
4088: logic_rules = rules['logical_consistency']['scoring']
```

```
4089:     has_docstring_gt_50 = len(docstring) > 50
4090:     has_docstring_gt_20 = len(docstring) > 20
4091:     has_returns_doc = 'return' in docstring.lower()
4092:     has_params_doc = 'param' in docstring.lower() or 'arg' in docstring.lower()
4093:
4094:     if has_docstring_gt_50 and has_returns_doc and has_params_doc:
4095:         logical_score = logic_rules['complete_documentation']['score']
4096:     elif has_docstring_gt_20:
4097:         logical_score = logic_rules['partial_documentation']['score']
4098:     else:
4099:         logical_score = logic_rules['minimal_documentation']['score']
4100:
4101:     # Component 3: Appropriate assumptions
4102:     assumption_rules = rules['appropriate_assumptions']['scoring']
4103:     assumption_keywords = assumption_rules['assumptions_documented']['keywords']
4104:     assumption_matches = [kw for kw in assumption_keywords if kw in docstring.lower()]
4105:
4106:     if len(assumption_matches) > 0:
4107:         assumptions_score = assumption_rules['assumptions_documented']['score']
4108:     else:
4109:         assumptions_score = assumption_rules['implicit_assumptions']['score']
4110:
4111:     # Weighted combination per rubric
4112:     b_theory = (
4113:         weights['grounded_in_valid_statistics'] * stat_score +
4114:         weights['logical_consistency'] * logical_score +
4115:         weights['appropriate_assumptions'] * assumptions_score
4116:     )
4117:
4118:     # Machine-readable evidence
4119:     evidence = {
4120:         "formula": "b_theory = 0.4*stat + 0.3*logic + 0.3*assumptions",
4121:         "components": {
4122:             "grounded_in_valid_statistics": {
4123:                 "weight": weights['grounded_in_valid_statistics'],
4124:                 "score": stat_score,
4125:                 "matched_keywords": stat_matches,
4126:                 "keyword_count": len(stat_matches),
4127:                 "rule_applied": "has_bayesian_or_statistical_model" if len(stat_matches) >= 3
4128:                               else "has_some_statistical_grounding" if len(stat_matches) >= 1
4129:                               else "no_statistical_grounding"
4130:             },
4131:             "logical_consistency": {
4132:                 "weight": weights['logical_consistency'],
4133:                 "score": logical_score,
4134:                 "docstring_length": len(docstring),
4135:                 "has_returns_doc": has_returns_doc,
4136:                 "has_params_doc": has_params_doc,
4137:                 "rule_applied": "complete_documentation" if (has_docstring_gt_50 and has_returns_doc and has_params_doc)
4138:                               else "partial_documentation" if has_docstring_gt_20
4139:                               else "minimal_documentation"
4140:             },
4141:             "appropriate_assumptions": {
4142:                 "weight": weights['appropriate_assumptions'],
4143:                 "score": assumptions_score,
4144:                 "matched_keywords": assumption_matches,
```

```
4145:             "rule_applied": "assumptions_documented" if assumption_matches else "implicit_assumptions"
4146:         }
4147:     },
4148:     "final_score": round(b_theory, 3),
4149:     "rubric_version": rubric['_metadata']['version']
4150:   }
4151:
4152:   return round(b_theory, 3), evidence
4153:
4154:
4155: def get_method_source(repo_root: Path, file_path: str, start_line: int) -> str:
4156: """
4157: Attempt to read the source code of a method.
4158: This is a heuristic extraction based on indentation.
4159: """
4160: try:
4161:     full_path = repo_root / "src" / file_path
4162:     if not full_path.exists():
4163:         return ""
4164:
4165:     with open(full_path, 'r') as f:
4166:         lines = f.readlines()
4167:
4168:     if start_line < 1 or start_line > len(lines):
4169:         return ""
4170:
4171:     # Adjust for 0-indexing
4172:     start_idx = start_line - 1
4173:     method_def = lines[start_idx]
4174:
4175:     # Determine indentation of the def line
4176:     indentation = len(method_def) - len(method_def.lstrip())
4177:
4178:     source_lines = [method_def]
4179:
4180:     for i in range(start_idx + 1, len(lines)):
4181:         line = lines[i]
4182:         if not line.strip(): # Keep empty lines
4183:             source_lines.append(line)
4184:             continue
4185:
4186:         current_indent = len(line) - len(line.lstrip())
4187:         if current_indent <= indentation:
4188:             break # End of method
4189:         source_lines.append(line)
4190:
4191:     return "".join(source_lines)
4192: except Exception:
4193:     return ""
4194:
4195: def compute_b_impl(method_info: Dict[str, Any], repo_root: Path, rubric: Dict[str, Any]) -> Tuple[float, Dict]:
4196: """
4197: Compute bImpl: implementation quality
4198:
4199: Uses machine-readable rules from rubric config
4200: """
```

```

4201:     signature = method_info.get('signature', '')
4202:     docstring = method_info.get('docstring', '') or ''
4203:     input_params = method_info.get('input_parameters', [])
4204:     return_type = method_info.get('return_type', None)
4205:     file_path = method_info.get('file_path', '')
4206:     line_number = method_info.get('line_number', 0)
4207:
4208:     # Load rubric rules
4209:     b_impl_config = rubric['b_impl']
4210:     weights = b_impl_config['weights']
4211:     rules = b_impl_config['rules']
4212:
4213:     # Component 1: Test coverage (conservative default)
4214:     test_rules = rules['test_coverage']['scoring']
4215:     test_score = test_rules['no_test_evidence']['score'] # Conservative default
4216:
4217:     # Component 2: Type annotations (use formula from rubric)
4218:     params_with_types = sum(1 for p in input_params if p.get('type_hint'))
4219:     total_params = max(len(input_params), 1)
4220:     has_return_type = return_type is not None and return_type != ''
4221:     # Formula: (typed_params / total_params) * 0.7 + (0.3 if has_return_type else 0)
4222:     type_score = (params_with_types / total_params * 0.7) + (0.3 if has_return_type else 0)
4223:
4224:     # Component 3: Error handling (detect try/except)
4225:     error_rules = rules['error_handling']['scoring']
4226:
4227:     # Try to read source to find try/except
4228:     source_code = get_method_source(repo_root, file_path, line_number)
4229:     has_try_except = "try:" in source_code and "except" in source_code
4230:
4231:     if has_try_except:
4232:         error_score = error_rules['comprehensive_handling']['score']
4233:         error_rule = "comprehensive_handling"
4234:     else:
4235:         error_score = error_rules['minimal_handling']['score']
4236:         error_rule = "minimal_handling"
4237:
4238:     # Component 4: Documentation (use formula from rubric)
4239:     doc_length = len(docstring)
4240:     has_description = doc_length > 50
4241:     has_params_doc = 'param' in docstring.lower() or 'arg' in docstring.lower()
4242:     has_returns_doc = 'return' in docstring.lower()
4243:     has_examples = 'example' in docstring.lower()
4244:     # Formula: (0.4 if doc_length > 50 else 0.1) + (0.3 if has_params_doc else 0) + (0.2 if has_returns_doc else 0) + (0.1 if has_examples else 0)
4245:     doc_score = (
4246:         (0.4 if has_description else 0.1) +
4247:         (0.3 if has_params_doc else 0) +
4248:         (0.2 if has_returns_doc else 0) +
4249:         (0.1 if has_examples else 0)
4250:     )
4251:
4252:     # Weighted combination per rubric
4253:     b_impl = (
4254:         weights['test_coverage'] * test_score +
4255:         weights['type_annotations'] * type_score +
4256:         weights['error_handling'] * error_score +

```

```

4257:         weights['documentation'] * doc_score
4258:     )
4259:
4260:     # Machine-readable evidence
4261:     evidence = {
4262:         "formula": "b_impl = 0.35*test + 0.25*type + 0.25*error + 0.15*doc",
4263:         "components": {
4264:             "test_coverage": {
4265:                 "weight": weights['test_coverage'],
4266:                 "score": test_score,
4267:                 "rule_applied": "no_test_evidence",
4268:                 "note": "Conservative default until measured"
4269:             },
4270:             "type_annotations": {
4271:                 "weight": weights['type_annotations'],
4272:                 "score": round(type_score, 3),
4273:                 "formula": "(typed_params / total_params) * 0.7 + (0.3 if has_return_type else 0)",
4274:                 "details": {
4275:                     "typed_params": params_with_types,
4276:                     "total_params": total_params,
4277:                     "has_return_type": has_return_type
4278:                 }
4279:             },
4280:             "error_handling": {
4281:                 "weight": weights['error_handling'],
4282:                 "score": error_score,
4283:                 "rule_applied": error_rule,
4284:                 "has_try_except": has_try_except
4285:             },
4286:             "documentation": {
4287:                 "weight": weights['documentation'],
4288:                 "score": round(doc_score, 3),
4289:                 "formula": "weighted_sum(desc, params, returns, examples)",
4290:                 "details": {
4291:                     "doc_length": doc_length,
4292:                     "has_params": has_params_doc,
4293:                     "has_returns": has_returns_doc,
4294:                     "has_examples": has_examples
4295:                 }
4296:             }
4297:         },
4298:         "final_score": round(b_impl, 3),
4299:         "rubric_version": rubric['_metadata']['version']
4300:     }
4301:
4302:     return round(b_impl, 3), evidence
4303:
4304:
4305: def compute_b_deploy(method_info: Dict[str, Any], rubric: Dict[str, Any]) -> Tuple[float, Dict]:
4306:     """
4307:     Compute b_deploy: deployment maturity
4308:
4309:     Uses machine-readable rules from rubric config
4310:     """
4311:     layer = method_info.get('layer', 'unknown')
4312:

```

```
4313:     # Load rubric rules
4314:     b_deploy_config = rubric['b_deploy']
4315:     weights = b_deploy_config['weights']
4316:     rules = b_deploy_config['rules']
4317:
4318:     # Get layer maturity baseline from rubric
4319:     layer_maturity_map = rules['layer_maturity_baseline']['scoring']
4320:     base_maturity = layer_maturity_map.get(layer, layer_maturity_map['unknown'])
4321:
4322:     # Apply formulas from rubric
4323:     # validation_runs: layer_maturity_baseline * 0.8
4324:     validation_score = base_maturity * 0.8
4325:
4326:     # stability_coefficient: layer_maturity_baseline * 0.9
4327:     stability_score = base_maturity * 0.9
4328:
4329:     # failure_rate: layer_maturity_baseline * 0.85
4330:     failure_score = base_maturity * 0.85
4331:
4332:     # Weighted combination per rubric
4333:     b_deploy = (
4334:         weights['validation_runs'] * validation_score +
4335:         weights['stability_coefficient'] * stability_score +
4336:         weights['failure_rate'] * failure_score
4337:     )
4338:
4339:     # Machine-readable evidence
4340:     evidence = {
4341:         "formula": "b_deploy = 0.4*validation + 0.35*stability + 0.25*failure",
4342:         "components": {
4343:             "layer_maturity_baseline": {
4344:                 "layer": layer,
4345:                 "baseline_score": base_maturity,
4346:                 "source": "rubric layer_maturity_baseline mapping"
4347:             },
4348:             "validation_runs": {
4349:                 "weight": weights['validation_runs'],
4350:                 "score": round(validation_score, 3),
4351:                 "formula": "layer_maturity_baseline * 0.8",
4352:                 "computation": f"{base_maturity} * 0.8 = {round(validation_score, 3)}"
4353:             },
4354:             "stability_coefficient": {
4355:                 "weight": weights['stability_coefficient'],
4356:                 "score": round(stability_score, 3),
4357:                 "formula": "layer_maturity_baseline * 0.9",
4358:                 "computation": f"{base_maturity} * 0.9 = {round(stability_score, 3)}"
4359:             },
4360:             "failure_rate": {
4361:                 "weight": weights['failure_rate'],
4362:                 "score": round(failure_score, 3),
4363:                 "formula": "layer_maturity_baseline * 0.85",
4364:                 "computation": f"{base_maturity} * 0.85 = {round(failure_score, 3)}"
4365:             }
4366:         },
4367:         "final_score": round(b_deploy, 3),
4368:         "rubric_version": rubric['_metadata']['version']
```

```
4369:     }
4370:
4371:     return round(b_deploy, 3), evidence
4372:
4373:
4374: def triage_and_calibrate_method(method_info: Dict[str, Any], repo_root: Path, rubric: Dict[str, Any]) -> Dict[str, Any]:
4375:     """
4376:     Full triage and calibration for one method using rubric.
4377:
4378:     Returns calibration entry for intrinsic_calibration.json
4379:     """
4380:     canonical_name = method_info.get('canonical_name', '')
4381:
4382:     # Pass 1: Requires calibration?
4383:     requires_cal, reason, triage_evidence = triage_pass1_requires_calibration(method_info, rubric)
4384:
4385:     if not requires_cal:
4386:         # Excluded method
4387:         return {
4388:             "method_id": canonical_name,
4389:             "calibration_status": "excluded",
4390:             "reason": reason,
4391:             "triage_evidence": triage_evidence,
4392:             "layer": method_info.get('layer', 'unknown'),
4393:             "last_updated": datetime.now(timezone.utc).isoformat(),
4394:             "approved_by": "automated_triage",
4395:             "rubric_version": rubric['_metadata']['version']
4396:         }
4397:
4398:     # Pass 2: Compute intrinsic calibration scores using rubric
4399:     b_theory, theory_evidence = compute_b_theory(method_info, repo_root, rubric)
4400:     b_impl, impl_evidence = compute_b_impl(method_info, repo_root, rubric)
4401:     b_deploy, deploy_evidence = compute_b_deploy(method_info, rubric)
4402:
4403:     # Pass 3: Create calibration profile with machine-readable evidence
4404:     return {
4405:         "method_id": canonical_name,
4406:         "b_theory": b_theory,
4407:         "b_impl": b_impl,
4408:         "b_deploy": b_deploy,
4409:         "evidence": {
4410:             "triage_decision": triage_evidence,
4411:             "triage_reason": reason,
4412:             "b_theory_computation": theory_evidence,
4413:             "b_impl_computation": impl_evidence,
4414:             "b_deploy_computation": deploy_evidence
4415:         },
4416:         "calibration_status": "computed",
4417:         "layer": method_info.get('layer', 'unknown'),
4418:         "last_updated": datetime.now(timezone.utc).isoformat(),
4419:         "approved_by": "automated_triage_with_rubric",
4420:         "rubric_version": rubric['_metadata']['version']
4421:     }
4422:
4423:
4424: def main():
```

```
4425:     """Execute rigorous method-by-method triage using machine-readable rubric"""
4426:     repo_root = Path(__file__).resolve().parents[4]
4427:     catalogue_path = repo_root / "config" / "canonical_method_catalogue_v2.json"
4428:     rubric_path = Path(__file__).resolve().parent / "intrinsic_calibration_rubric.json" # Same directory as script
4429:     output_path = repo_root / "config" / "intrinsic_calibration.json"
4430:
4431:     print("Loading machine-readable rubric...")
4432:     rubric = load_json(rubric_path)
4433:     print(f" Rubric version: {rubric['_metadata']['version']} ")
4434:
4435:     print("Loading canonical method catalogue...")
4436:     catalogue = load_json(catalogue_path)
4437:     print(f" Total methods in catalogue: {len(catalogue)} ")
4438:
4439:     print("Loading current intrinsic calibrations...")
4440:     if output_path.exists():
4441:         intrinsic = load_json(output_path)
4442:     else:
4443:         intrinsic = {}
4444:
4445:     # Get existing calibrations (keep manually curated ones)
4446:     existing_methods = {}
4447:     for method_id, profile in intrinsic.items():
4448:         if not method_id.startswith("_"):
4449:             # Keep if approved_by indicates manual curation
4450:             if "system_architect" in profile.get("approved_by", ""):
4451:                 existing_methods[method_id] = profile
4452:
4453:     print(f"Preserving {len(existing_methods)} manually curated calibrations")
4454:
4455:     #Process ALL catalogue methods (flat array structure)
4456:     all_methods = {}
4457:     for method_info in catalogue:
4458:         unique_id = method_info.get("unique_id", "")
4459:         if unique_id:
4460:             # Add method_name field from canonical_name for compatibility
4461:             method_info['method_name'] = method_info.get('canonical_name', '')
4462:             all_methods[unique_id] = method_info
4463:
4464:     print(f"\nProcessing {len(all_methods)} methods with rubric-based triage...")
4465:     print("=" * 80)
4466:
4467:     processed = 0
4468:     calibrated = 0
4469:     excluded = 0
4470:
4471:     new_methods = {}
4472:
4473:     for method_id, method_info in sorted(all_methods.items()):
4474:         # Keep existing manual calibrations
4475:         if method_id in existing_methods:
4476:             new_methods[method_id] = existing_methods[method_id]
4477:             calibrated += 1
4478:         else:
4479:             # Apply triage process with rubric
4480:             calibration_entry = triage_and_calibrate_method(method_info, repo_root, rubric)
```

```
4481:         new_methods[method_id] = calibration_entry
4482:
4483:         if calibration_entry.get("calibration_status") == "excluded":
4484:             excluded += 1
4485:         else:
4486:             calibrated += 1
4487:
4488:     processed += 1
4489:     if processed % 100 == 0:
4490:         print(f"  Processed {processed}/{len(all_methods)} methods...")
4491:
4492:     # Update intrinsic calibration file
4493:     output_data = {
4494:         "_metadata": {
4495:             "version": "2.0.0",
4496:             "generated": datetime.now(timezone.utc).isoformat(),
4497:             "total_methods": len(all_methods),
4498:             "computed_methods": calibrated,
4499:             "excluded_methods": excluded,
4500:             "coverage_percent": round((calibrated / len(all_methods)) * 100, 2),
4501:             "rubric_version": rubric['_metadata']['version'],
4502:             "rubric_reference": "src/farfan_pipeline/core/calibration/intrinsic_calibration_rubric.json",
4503:             "methodology": "Machine-readable rubric with traceable evidence",
4504:             "reproducibility": "All scores can be regenerated from rubric + catalogue"
4505:         }
4506:     }
4507:     output_data.update(new_methods)
4508:
4509:     print(f"\nSaving intrinsic_calibration.json...")
4510:     save_json(output_path, output_data)
4511:
4512:     print("\n" + "=" * 80)
4513:     print("RIGOROUS TRIAGE COMPLETE")
4514:     print("=" * 80)
4515:     print(f"Total methods processed: {len(all_methods)}")
4516:     print(f"Methods calibrated: {calibrated}")
4517:     print(f"Methods excluded: {excluded}")
4518:     print(f"Coverage: {calibrated/len(all_methods)*100:.2f}%")
4519:     print(f"Rubric version: {rubric['_metadata']['version']}")
4520:     print("\n" * 223 + "Every method analyzed using machine-readable rubric")
4521:     print(" " * 223 + "All scores traceable with explicit formulas and evidence")
4522:     print(" " * 223 + "Scores are reproducible from rubric + catalog")
4523:
4524:     return 0
4525:
4526:
4527: if __name__ == "__main__":
4528:     sys.exit(main())
4529:
4530:
4531:
4532: =====
4533: FILE: src/farfan_pipeline/core/calibration/unit_layer.py
4534: =====
4535:
4536: """
```

```
4537: Unit Layer (@u) - PRODUCTION IMPLEMENTATION.
4538:
4539: Evaluates PDT quality through 4 components: S, M, I, P.
4540: """
4541: import logging
4542: from farfan_pipeline.core.calibration.config import UnitLayerConfig
4543: from farfan_pipeline.core.calibration.pdt_structure import PDTStructure
4544: from farfan_pipeline.core.calibration.data_structures import LayerID, LayerScore
4545:
4546: logger = logging.getLogger(__name__)
4547:
4548:
4549: class UnitLayerEvaluator:
4550:     """
4551:         Evaluates Unit Layer (@u) - PDT quality.
4552:
4553:     PRODUCTION IMPLEMENTATION - All scores are data-driven.
4554:     """
4555:
4556:     # Mandatory blocks required for PDT compliance
4557:     MANDATORY_BLOCKS = ["DiagnÃstico", "Parte EstratÃgica", "PPI", "Seguimiento"]
4558:
4559:     def __init__(self, config: UnitLayerConfig):
4560:         self.config = config
4561:
4562:     def evaluate(self, pdt: PDTStructure) -> LayerScore:
4563:         """
4564:             Production implementation - computes S, M, I, P from PDT data.
4565:
4566:             THIS IS NOT A STUB - all scores are data-driven.
4567:             """
4568:             logger.info("unit_layer_evaluation_start", extra={"tokens": pdt.total_tokens})
4569:
4570:             # Step 1: Compute S (Structural Compliance)
4571:             S = self._compute_structural_compliance(pdt)
4572:             logger.info("S_computed", extra={"S": S})
4573:
4574:             # Step 2: Check hard gate for S
4575:             if S < self.config.min_structural_compliance:
4576:                 return LayerScore(
4577:                     layer=LayerID.UNIT,
4578:                     score=0.0,
4579:                     components={"S": S, "gate_failure": "structural"},
4580:                     rationale=f"HARD GATE: S={S:.2f} < {self.config.min_structural_compliance}",
4581:                     metadata={"gate": "structural", "threshold": self.config.min_structural_compliance}
4582:                 )
4583:
4584:             # Step 3: Compute M (Mandatory Sections)
4585:             M = self._compute_mandatory_sections(pdt)
4586:             logger.info("M_computed", extra={"M": M})
4587:
4588:             # Step 4: Compute I (Indicator Quality)
4589:             I_components = self._compute_indicator_quality(pdt)
4590:             I = I_components["I_total"]
4591:             logger.info("I_computed", extra={"I": I})
4592:
```

```
4593:     # Step 5: Check hard gate for I_struct
4594:     if I_components["I_struct"] < self.config.i_struct_hard_gate:
4595:         return LayerScore(
4596:             layer=LayerID.UNIT,
4597:             score=0.0,
4598:             components={"S": S, "M": M, "I_struct": I_components["I_struct"]},
4599:             rationale=f"HARD GATE: I_struct={I_components['I_struct']:.2f} < {self.config.i_struct_hard_gate}",
4600:             metadata={"gate": "indicator_structure"}
4601:         )
4602:
4603:     # Step 6: Compute P (PPI Completeness)
4604:     P_components = self._compute_ppi_completeness(pdt)
4605:     P = P_components["P_total"]
4606:     logger.info("P_computed", extra={"P": P})
4607:
4608:     # Step 7: Check hard gates for PPI
4609:     if self.config.require_ppi_presence and not pdt.ppi_matrix_present:
4610:         return LayerScore(
4611:             layer=LayerID.UNIT,
4612:             score=0.0,
4613:             components={"S": S, "M": M, "I": I, "gate_failure": "ppi_presence"},
4614:             rationale="HARD GATE: PPI required but not present",
4615:             metadata={"gate": "ppi_presence"}
4616:         )
4617:
4618:     if self.config.require_indicator_matrix and not pdt.indicator_matrix_present:
4619:         return LayerScore(
4620:             layer=LayerID.UNIT,
4621:             score=0.0,
4622:             components={"S": S, "M": M, "I": I, "P": P, "gate_failure": "indicator_matrix"},
4623:             rationale="HARD GATE: Indicator matrix required but not present",
4624:             metadata={"gate": "indicator_matrix"}
4625:         )
4626:
4627:     # Step 8: Aggregate
4628:     U_base = self._aggregate_components(S, M, I, P)
4629:     logger.info("U_base_computed", extra={"U_base": U_base})
4630:
4631:     # Step 9: Anti-gaming
4632:     gaming_penalty = self._compute_gaming_penalty(pdt)
4633:     U_final = max(0.0, U_base - gaming_penalty)
4634:
4635:     # Step 10: Quality level
4636:     if U_final >= 0.85:
4637:         quality = "sobresaliente"
4638:     elif U_final >= 0.7:
4639:         quality = "robusto"
4640:     elif U_final >= 0.5:
4641:         quality = "mÁ-nimo"
4642:     else:
4643:         quality = "insuficiente"
4644:
4645:     return LayerScore(
4646:         layer=LayerID.UNIT,
4647:         score=U_final,
4648:         components={"S": S, "M": M, "I": I, "P": P, "U_base": U_base, "penalty": gaming_penalty},
```

```
4649:         rationale=f"Unit quality: {quality} (S={S:.2f}, M={M:.2f}, I={I:.2f}, P={P:.2f})",
4650:         metadata={"quality_level": quality, "aggregation": self.config.aggregation_type}
4651:     )
4652:
4653:     def _compute_structural_compliance(self, pdt: PDTStructure) -> float:
4654:         """Compute S = w_block*B_cov + w_hierarchy*H + w_order*O."""
4655:         # Block coverage
4656:         blocks_found = sum(
4657:             1 for block in self.MANDATORY_BLOCKS
4658:             if block in pdt.blocks_found
4659:             and pdt.blocks_found[block].get("tokens", 0) >= self.config.min_block_tokens
4660:             and pdt.blocks_found[block].get("numbers_count", 0) >= self.config.min_block_numbers
4661:         )
4662:         B_cov = blocks_found / len(self.MANDATORY_BLOCKS)
4663:
4664:         # Hierarchy score
4665:         if not pdt.headers:
4666:             H = 0.0
4667:         else:
4668:             valid = sum(1 for h in pdt.headers if h.get("valid_numbering"))
4669:             ratio = valid / len(pdt.headers)
4670:             if ratio >= self.config.hierarchy_excellent_threshold:
4671:                 H = 1.0
4672:             elif ratio >= self.config.hierarchy_acceptable_threshold:
4673:                 H = 0.5
4674:             else:
4675:                 H = 0.0
4676:
4677:         # Order score - count inversions in block_sequence vs expected
4678:         expected = ["DiagnÃ³stico", "Parte EstratÃ©gica", "PPI", "Seguimiento"]
4679:         inversions = 0
4680:         if pdt.block_sequence:
4681:             # Find positions of blocks in actual sequence
4682:             positions = {}
4683:             for i, block in enumerate(pdt.block_sequence):
4684:                 if block in expected:
4685:                     positions[block] = i
4686:
4687:             # Count inversions (pairs out of order)
4688:             for i, block1 in enumerate(expected):
4689:                 if block1 not in positions:
4690:                     continue
4691:                 for block2 in expected[i+1:]:
4692:                     if block2 not in positions:
4693:                         continue
4694:                     if positions[block1] > positions[block2]:
4695:                         inversions += 1
4696:
4697:             O = 1.0 if inversions == 0 else (0.5 if inversions == 1 else 0.0)
4698:
4699:             S = (self.config.w_block_coverage * B_cov +
4700:                  self.config.w_hierarchy * H +
4701:                  self.config.w_order * O)
4702:
4703:         return S
4704:
```

```
4705:     def _compute_mandatory_sections(self, pdt: PDTStructure) -> float:
4706:         """Compute M = weighted average of section completeness."""
4707:         # Section requirements (from config)
4708:         requirements = {
4709:             "DiagnÃ³stico": {
4710:                 "min_tokens": self.config.diagnostico_min_tokens,
4711:                 "min_keywords": self.config.diagnostico_min_keywords,
4712:                 "min_numbers": self.config.diagnostico_min_numbers,
4713:                 "min_sources": self.config.diagnostico_min_sources,
4714:                 "weight": self.config.critical_sections_weight, # Critical section
4715:             },
4716:             "Parte Estrategica": {
4717:                 "min_tokens": self.config.estategica_min_tokens,
4718:                 "min_keywords": self.config.estategica_min_keywords,
4719:                 "min_numbers": self.config.estategica_min_numbers,
4720:                 "weight": self.config.critical_sections_weight, # Critical section
4721:             },
4722:             "PPI": {
4723:                 "min_tokens": self.config.ppi_section_min_tokens,
4724:                 "min_keywords": self.config.ppi_section_min_keywords,
4725:                 "min_numbers": self.config.ppi_section_min_numbers,
4726:                 "weight": self.config.critical_sections_weight, # Critical section
4727:             },
4728:             "Seguimiento": {
4729:                 "min_tokens": self.config.seguimiento_min_tokens,
4730:                 "min_keywords": self.config.seguimiento_min_keywords,
4731:                 "min_numbers": self.config.seguimiento_min_numbers,
4732:                 "weight": 1.0,
4733:             },
4734:             "Marco Normativo": {
4735:                 "min_tokens": self.config.marco_normativo_min_tokens,
4736:                 "min_keywords": self.config.marco_normativo_min_keywords,
4737:                 "weight": 1.0,
4738:             }
4739:         }
4740:
4741:         total_weight = 0.0
4742:         weighted_score = 0.0
4743:
4744:         for section_name, reqs in requirements.items():
4745:             section_data = pdt.sections_found.get(section_name, {})
4746:
4747:             if not section_data.get("present", False):
4748:                 # Missing section gets 0
4749:                 score = 0.0
4750:
4751:             else:
4752:                 # Check all requirements
4753:                 checks_passed = 0
4754:                 checks_total = 0
4755:
4756:                 if "min_tokens" in reqs:
4757:                     checks_total += 1
4758:                     if section_data.get("token_count", 0) >= reqs["min_tokens"]:
4759:                         checks_passed += 1
4760:
4761:                 if "min_keywords" in reqs:
```

```

4761:             checks_total += 1
4762:             if section_data.get("keyword_matches", 0) >= reqs["min_keywords"]:
4763:                 checks_passed += 1
4764:
4765:             if "min_numbers" in reqs:
4766:                 checks_total += 1
4767:                 if section_data.get("number_count", 0) >= reqs["min_numbers"]:
4768:                     checks_passed += 1
4769:
4770:             if "min_sources" in reqs:
4771:                 checks_total += 1
4772:                 if section_data.get("sources_found", 0) >= reqs["min_sources"]:
4773:                     checks_passed += 1
4774:
4775:             score = checks_passed / checks_total if checks_total > 0 else 0.0
4776:
4777:             weight = reqs.get("weight", 1.0)
4778:             weighted_score += score * weight
4779:             total_weight += weight
4780:
4781:             M = weighted_score / total_weight if total_weight > 0 else 0.0
4782:             return M
4783:
4784:     def _compute_indicator_quality(self, pdt: PDTStructure) -> dict:
4785:         """Compute I = w_struct*I_struct + w_link*I_link + w_logic*I_logic."""
4786:         if not pdt.indicator_matrix_present or not pdt.indicator_rows:
4787:             logger.warning("indicator_matrix_absent", extra={"I": 0.0})
4788:             return {
4789:                 "I_struct": 0.0,
4790:                 "I_link": 0.0,
4791:                 "I_logic": 0.0,
4792:                 "I_total": 0.0
4793:             }
4794:
4795:         # I_struct: Field completeness
4796:         critical_fields = ["Tipo", "LÃnea EstratÃgica", "Programa", "LÃnea Base",
4797:                            "Meta Cuatrienio", "Fuente", "Unidad Medida"]
4798:         optional_fields = ["Ã±o LB", "CÃ³digo MGA"]
4799:
4800:         total_struct_score = 0.0
4801:         for row in pdt.indicator_rows:
4802:             critical_present = sum(1 for f in critical_fields if row.get(f))
4803:             optional_present = sum(1 for f in optional_fields if row.get(f))
4804:
4805:             # Penalize placeholders
4806:             placeholder_count = sum(
4807:                 1 for f in critical_fields
4808:                     if row.get(f) in ["S/D", "N/A", "TBD", ""])
4809:
4810:
4811:             critical_score = critical_present / len(critical_fields)
4812:             optional_score = optional_present / len(optional_fields)
4813:             placeholder_penalty = (placeholder_count / len(critical_fields)) * self.config.i_placeholder_penalty_multiplier
4814:
4815:             row_score = (critical_score * self.config.i_critical_fields_weight + optional_score) / (self.config.i_critical_fields_weight + 1)
4816:             row_score = max(0.0, row_score - placeholder_penalty)

```

```

4817:         total_struct_score += row_score
4818:
4819:     I_struct = total_struct_score / len(pdt.indicator_rows)
4820:
4821:     # I_link: Traceability (fuzzy matching between indicators and strategic lines)
4822:     linked_count = 0
4823:     for row in pdt.indicator_rows:
4824:         programa = row.get("Programa", "")
4825:         linea = row.get("LÃ±ea EstratÃ©gica", "")
4826:         if programa and linea:
4827:             # Simplified: check if they share significant words
4828:             prog_words = set(programa.lower().split())
4829:             linea_words = set(linea.lower().split())
4830:             if len(prog_words & linea_words) >= 2: # At least 2 words in common
4831:                 linked_count += 1
4832:
4833:     I_link = linked_count / len(pdt.indicator_rows)
4834:
4835:     # I_logic: Year coherence
4836:     logic_violations = 0
4837:     for row in pdt.indicator_rows:
4838:         year_lb = row.get("AÃ±o LB")
4839:
4840:         if year_lb is not None:
4841:             try:
4842:                 year_lb_int = int(year_lb)
4843:                 if not (self.config.i_valid_lb_year_min <= year_lb_int <= self.config.i_valid_lb_year_max):
4844:                     logic_violations += 1
4845:             except (ValueError, TypeError):
4846:                 # Invalid year format counts as violation
4847:                 logic_violations += 1
4848:
4849:     I_logic = 1.0 - (logic_violations / len(pdt.indicator_rows))
4850:
4851:     # Aggregate
4852:     I_total = (self.config.w_i_struct * I_struct +
4853:                self.config.w_i_link * I_link +
4854:                self.config.w_i_logic * I_logic)
4855:
4856:     return {
4857:         "I_struct": I_struct,
4858:         "I_link": I_link,
4859:         "I_logic": I_logic,
4860:         "I_total": I_total
4861:     }
4862:
4863:     def _compute_ppi_completeness(self, pdt: PDTStructure) -> dict:
4864:         """Compute P = w_presence•P_presence + w_struct•P_struct + w_consistency•P_consistency."""
4865:         # P_presence
4866:         P_presence = 1.0 if pdt.ppi_matrix_present else 0.0
4867:
4868:         if not pdt.ppi_matrix_present or not pdt.ppi_rows:
4869:             return {
4870:                 "P_presence": P_presence,
4871:                 "P_struct": 0.0,
4872:                 "P_consistency": 0.0,

```

```

4873:             "P_total": P_presence * self.config.w_p_presence
4874:         }
4875:
4876:     # P_struct: Non-zero rows
4877:     nonzero_rows = sum(
4878:         1 for row in pdt.ppi_rows
4879:         if row.get("Costo Total", 0) > 0
4880:     )
4881:     P_struct = nonzero_rows / len(pdt.ppi_rows)
4882:
4883:     # P_consistency: Accounting closure
4884:     violations = 0
4885:     for row in pdt.ppi_rows:
4886:         costo_total = row.get("Costo Total", 0)
4887:
4888:         # Check temporal sum
4889:         temporal_sum = sum(row.get(str(year), 0) for year in range(2024, 2028))
4890:         if abs(temporal_sum - costo_total) > costo_total * self.config.p_accounting_tolerance:
4891:             violations += 1
4892:
4893:         # Check source sum
4894:         source_sum = (row.get("SGP", 0) + row.get("SGR", 0) +
4895:                       row.get("Propios", 0) + row.get("Otras", 0))
4896:         if abs(source_sum - costo_total) > costo_total * self.config.p_accounting_tolerance:
4897:             violations += 1
4898:
4899:     P_consistency = 1.0 - (violations / (len(pdt.ppi_rows) * 2)) # 2 checks per row
4900:
4901:     # Aggregate
4902:     P_total = (self.config.w_p_presence * P_presence +
4903:                 self.config.w_p_structure * P_struct +
4904:                 self.config.w_p_consistency * P_consistency)
4905:
4906:     return {
4907:         "P_presence": P_presence,
4908:         "P_struct": P_struct,
4909:         "P_consistency": P_consistency,
4910:         "P_total": P_total
4911:     }
4912:
4913: def _aggregate_components(self, S: float, M: float, I: float, P: float) -> float:
4914:     """Aggregate S, M, I, P using configured method."""
4915:     if self.config.aggregation_type == "geometric_mean":
4916:         # Geometric mean: (S*M*I*P)^(1/4)
4917:         product = S * M * I * P
4918:         return product ** 0.25
4919:     elif self.config.aggregation_type == "harmonic_mean":
4920:         # Harmonic mean: 4 / (1/S + 1/M + 1/I + 1/P)
4921:         if S == 0 or M == 0 or I == 0 or P == 0:
4922:             return 0.0
4923:         return 4.0 / (1.0/S + 1.0/M + 1.0/I + 1.0/P)
4924:     else: # weighted_average
4925:         return (self.config.w_S * S +
4926:                 self.config.w_M * M +
4927:                 self.config.w_I * I +
4928:                 self.config.w_P * P)

```

```
4929:  
4930:     def _compute_gaming_penalty(self, pdt: PDTStructure) -> float:  
4931:         """Compute anti-gaming penalties."""  
4932:         penalties = []  
4933:  
4934:         # Check placeholder ratio in indicators  
4935:         if pdt.indicator_matrix_present and pdt.indicator_rows:  
4936:             placeholder_count = 0  
4937:             total_fields = 0  
4938:             for row in pdt.indicator_rows:  
4939:                 for key, value in row.items():  
4940:                     total_fields += 1  
4941:                     if value in ["S/D", "N/A", "TBD", ""]:  
4942:                         placeholder_count += 1  
4943:  
4944:             placeholder_ratio = placeholder_count / total_fields if total_fields > 0 else 0  
4945:             if placeholder_ratio > self.config.max_placeholder_ratio:  
4946:                 penalty = (placeholder_ratio - self.config.max_placeholder_ratio) * 0.5  
4947:                 penalties.append(penalty)  
4948:  
4949:         # Check unique values in PPI costs  
4950:         if pdt.ppi_matrix_present and pdt.ppi_rows:  
4951:             costs = [row.get("Costo Total", 0) for row in pdt.ppi_rows]  
4952:             unique_costs = len(set(costs))  
4953:             unique_ratio = unique_costs / len(costs) if costs else 0  
4954:  
4955:             if unique_ratio < self.config.min_unique_values_ratio:  
4956:                 penalty = (self.config.min_unique_values_ratio - unique_ratio) * 0.3  
4957:                 penalties.append(penalty)  
4958:  
4959:         # Check number density in critical sections  
4960:         critical_sections = ["DiagnÃ³stico", "Parte EstratÃ©gica", "PPI"]  
4961:         for section in critical_sections:  
4962:             section_data = pdt.sections_found.get(section, {})  
4963:             if section_data.get("present"):  
4964:                 tokens = section_data.get("token_count", 0)  
4965:                 numbers = section_data.get("number_count", 0)  
4966:                 density = numbers / tokens if tokens > 0 else 0  
4967:  
4968:                 if density < self.config.min_number_density:  
4969:                     penalty = (self.config.min_number_density - density) * 0.2  
4970:                     penalties.append(penalty)  
4971:  
4972:         total_penalty = sum(penalties)  
4973:         return min(total_penalty, self.config.gaming_penalty_cap)  
4974:  
4975:  
4976:  
4977: ======  
4978: FILE: src/farfan_pipeline/core/calibration/validators.py  
4979: ======  
4980:  
4981: """  
4982: Three-Pillar Calibration System - Validation Functions  
4983:  
4984: This module implements validation checks for the calibration system
```

```
4985: as specified in the SUPERPROMPT Three-Pillar Calibration System.
4986:
4987: Spec compliance: Section 8 (Validation & Governance)
4988: """
4989:
4990: import json
4991: from pathlib import Path
4992: from typing import Dict, Any, Set, List, TYPE_CHECKING
4993:
4994: if TYPE_CHECKING:
4995:     from farfan_pipeline.core.calibration.data_structures import CalibrationCertificate
4996:
4997: from farfan_pipeline.core.calibration.data_structures import MethodRole, LayerType, REQUIRED_LAYERS
4998:
4999:
5000: class CalibrationValidator:
5001: """
5002:     Validation system for calibration configs and runtime checks.
5003:
5004:     Spec compliance: Section 8 (Validation & Governance)
5005: """
5006:
5007:     def __init__(self, config_dir: str = None):
5008:         """Initialize validator with config directory"""
5009:         if config_dir is None:
5010:             config_dir = Path(__file__).parent.parent / "config"
5011:         else:
5012:             config_dir = Path(config_dir)
5013:
5014:         self.config_dir = config_dir
5015:
5016:     def validate_layer_completeness(
5017:         self,
5018:         method_id: str,
5019:         role: MethodRole,
5020:         declared_layers: Set[LayerType]
5021:     ) -> tuple[bool, List[str]]:
5022: """
5023:     Validate that method declares all required layers for its role.
5024:
5025:     Spec compliance: Section 4 (Theorem 4.1 - No Silent Defaults)
5026:
5027:     Args:
5028:         method_id: Canonical method ID
5029:         role: Method role
5030:         declared_layers: Set of layers method declares
5031:
5032:     Returns:
5033:         (is_valid, error_messages)
5034: """
5035:     required = REQUIRED_LAYERS.get(role, set())
5036:     missing = required - declared_layers
5037:
5038:     if not missing:
5039:         return True, []
5040:
```

```
5041:         errors = [
5042:             f"Method {method_id} (role={role.value}) missing required layers: "
5043:             f"\n{[l.value for l in missing]}"
5044:         ]
5045:     return False, errors
5046:
5047:     def validate_fusion_weights(
5048:         self,
5049:         role_params: Dict[str, Any],
5050:         role_name: str
5051:     ) -> tuple[bool, List[str]]:
5052:         """
5053:             Validate fusion weight normalization and constraints.
5054:
5055:             Spec compliance: Section 5 (Fusion Operator Constraints)
5056:
5057:             Constraints:
5058:             1.  $a_{\text{layer}} \geq 0$  for all  $\text{layer}$ 
5059:             2.  $a_{\text{layer}} \leq 1$  for all  $(\text{layer}, k)$ 
5060:             3.  $\sum a_{\text{layer}} + \sum a_{\text{layer}k} = 1.0$ 
5061:
5062:             Args:
5063:                 role_params: Parameters from fusion_specification.json
5064:                 role_name: Role identifier
5065:
5066:             Returns:
5067:                 (is_valid, error_messages)
5068:             """
5069:         errors = []
5070:
5071:         linear_weights = role_params.get("linear_weights", {})
5072:         interaction_weights = role_params.get("interaction_weights", {})
5073:
5074:         # Check non-negativity
5075:         for layer, weight in linear_weights.items():
5076:             if weight < 0:
5077:                 errors.append(
5078:                     f"Role {role_name}: linear weight for {layer} is negative: {weight}"
5079:                 )
5080:
5081:         for pair, weight in interaction_weights.items():
5082:             if weight < 0:
5083:                 errors.append(
5084:                     f"Role {role_name}: interaction weight for {pair} is negative: {weight}"
5085:                 )
5086:
5087:         # Check normalization
5088:         linear_sum = sum(linear_weights.values())
5089:         interaction_sum = sum(interaction_weights.values())
5090:         total = linear_sum + interaction_sum
5091:
5092:         tolerance = 1e-9
5093:         if abs(total - 1.0) > tolerance:
5094:             errors.append(
5095:                 f"Role {role_name}: weights do not sum to 1.0. "
5096:                 f"Linear={linear_sum:.6f}, Interaction={interaction_sum:.6f}, "
```

```
5097:             f"Total={total:.6f}"
5098:         )
5099:
5100:     return len(errors) == 0, errors
5101:
5102:     def validate_anti_universality(
5103:         self,
5104:         method_config: Dict[str, Any],
5105:         method_id: str,
5106:         contextual_config: Dict[str, Any],
5107:         monolith: Dict[str, Any]
5108:     ) -> tuple[bool, List[str]]:
5109:         """
5110:             Validate anti-universality constraint.
5111:
5112:             Spec compliance: Section 3.4 (Anti-Universality Constraint)
5113:
5114:             No method may have maximal compatibility (1.0) with ALL Q, D, and P.
5115:
5116:             Args:
5117:                 method_config: Method configuration
5118:                 method_id: Canonical method ID
5119:                 contextual_config: Contextual parametrization config
5120:                 monolith: Questionnaire monolith
5121:
5122:             Returns:
5123:                 (is_valid, error_messages)
5124:             """
5125:             errors = []
5126:
5127:             # For now, ensure policy default is < 1.0 (we set it to 0.9 in config)
5128:             policy_areas = contextual_config.get("layer_policy", {}).get("policy_areas", {})
5129:             all_policies_maximal = True
5130:
5131:             for policy_id, policy_spec in policy_areas.items():
5132:                 if policy_spec.get("default_score", 0.0) < 0.99:
5133:                     all_policies_maximal = False
5134:                     break
5135:
5136:             if all_policies_maximal:
5137:                 errors.append(
5138:                     f"Method {method_id} violates anti-universality: "
5139:                     f"all policy areas have maximal (â\211¥0.99) compatibility"
5140:                 )
5141:
5142:             return len(errors) == 0, errors
5143:
5144:     def validate_intrinsic_calibration(
5145:         self,
5146:         config: Dict[str, Any]
5147:     ) -> tuple[bool, List[str]]:
5148:         """
5149:             Validate intrinsic_calibration.json structure and values.
5150:
5151:             Args:
5152:                 config: Loaded intrinsic_calibration.json
```

```
5153:
5154:     Returns:
5155:         (is_valid, error_messages)
5156:     """
5157:     errors = []
5158:
5159:     # Check weights sum to 1.0
5160:     weights = config.get("_base_weights", {})
5161:     weight_sum = weights.get("w_th", 0) + weights.get("w_imp", 0) + weights.get("w_dep", 0)
5162:
5163:     if abs(weight_sum - 1.0) > 1e-9:
5164:         errors.append(f"Base weights do not sum to 1.0: {weight_sum}")
5165:
5166:     # Check each method entry
5167:     methods = config.get("methods", {})
5168:     for method_id, method_data in methods.items():
5169:         if method_id.startswith("_"):
5170:             continue # Skip metadata
5171:
5172:         # Check required fields
5173:         for field in ["b_theory", "b_impl", "b_deploy"]:
5174:             if field not in method_data:
5175:                 errors.append(f"Method {method_id} missing field: {field}")
5176:                 continue
5177:
5178:             value = method_data[field]
5179:             if not (0.0 <= value <= 1.0):
5180:                 errors.append(
5181:                     f"Method {method_id} field {field} out of bounds: {value}"
5182:                 )
5183:
5184:     return len(errors) == 0, errors
5185:
5186: def validate_config_files(self) -> tuple[bool, List[str]]:
5187:     """
5188:     Validate all three pillar config files.
5189:
5190:     Spec compliance: Section 8 (CI / QA Rules)
5191:
5192:     This should be run in CI to ensure config integrity.
5193:
5194:     Returns:
5195:         (is_valid, error_messages)
5196:     """
5197:     all_errors = []
5198:
5199:     # Load configs
5200:     try:
5201:         with open(self.config_dir / "intrinsic_calibration.json") as f:
5202:             intrinsic = json.load(f)
5203:     except Exception as e:
5204:         all_errors.append(f"Failed to load intrinsic_calibration.json: {e}")
5205:         intrinsic = {}
5206:
5207:     # Validate contextual config exists (full validation TBD)
5208:     try:
```

```
5209:     contextual_path = self.config_dir / "contextual_parametrization.json"
5210:     if not contextual_path.exists():
5211:         all_errors.append("contextual_parametrization.json not found")
5212:     except Exception as e:
5213:         all_errors.append(f"Failed to check contextual_parametrization.json: {e}")
5214:
5215:     try:
5216:         with open(self.config_dir / "fusion_specification.json") as f:
5217:             fusion = json.load(f)
5218:     except Exception as e:
5219:         all_errors.append(f"Failed to load fusion_specification.json: {e}")
5220:         fusion = {}
5221:
5222:     # Validate intrinsic calibration
5223:     if intrinsic:
5224:         valid, errors = self.validate_intrinsic_calibration(intrinsic)
5225:         all_errors.extend(errors)
5226:
5227:     # Validate fusion weights for each role
5228:     if fusion:
5229:         role_params = fusion.get("role_fusion_parameters", {})
5230:         for role_name, params in role_params.items():
5231:             valid, errors = self.validate_fusion_weights(params, role_name)
5232:             all_errors.extend(errors)
5233:
5234:     return len(all_errors) == 0, all_errors
5235:
5236:     def validate_boundedness(
5237:         self,
5238:         layer_scores: Dict[str, float],
5239:         calibrated_score: float
5240:     ) -> tuple[bool, List[str]]:
5241:         """
5242:             Validate boundedness constraint: all scores in [0,1].
5243:
5244:             Spec compliance: Section 8 (P1. Boundedness)
5245:
5246:             Args:
5247:                 layer_scores: All layer scores
5248:                 calibrated_score: Final calibrated score
5249:
5250:             Returns:
5251:                 (is_valid, error_messages)
5252:             """
5253:             errors = []
5254:
5255:             # Check layer scores
5256:             for layer, score in layer_scores.items():
5257:                 if not (0.0 <= score <= 1.0):
5258:                     errors.append(f"Layer {layer} score out of bounds: {score}")
5259:
5260:             # Check calibrated score
5261:             if not (0.0 <= calibrated_score <= 1.0):
5262:                 errors.append(f"Calibrated score out of bounds: {calibrated_score}")
5263:
5264:             return len(errors) == 0, errors
```

```
5265:  
5266:  
5267: # Convenience functions  
5268: def validate_config_files(config_dir: str = None) -> tuple[bool, List[str]]:  
5269:     """  
5270:         Validate all calibration config files.  
5271:  
5272:         This should be called in CI/CD pipelines.  
5273:         """  
5274:         validator = CalibrationValidator(config_dir=config_dir)  
5275:         return validator.validate_config_files()  
5276:  
5277:  
5278: def validate_certificate(  
5279:     certificate: 'CalibrationCertificate'  
5280: ) -> tuple[bool, List[str]]:  
5281:     """  
5282:         Validate a calibration certificate.  
5283:  
5284:         Args:  
5285:             certificate: CalibrationCertificate to validate  
5286:  
5287:         Returns:  
5288:             (is_valid, error_messages)  
5289:         """  
5290:         validator = CalibrationValidator()  
5291:  
5292:         # Check boundedness  
5293:         valid, errors = validator.validate_boundedness(  
5294:             certificate.layer_scores,  
5295:             certificate.calibrated_score  
5296:         )  
5297:  
5298:         return valid, errors  
5299:  
5300:  
5301:  
5302: =====  
5303: FILE: src/farfan_pipeline/core/dependency_lockdown.py  
5304: =====  
5305:  
5306: """Dependency lockdown enforcement to prevent magic downloads and hidden behavior.  
5307:  
5308: This module enforces explicit dependency management by:  
5309: 1. Checking if online model downloads are allowed via HF_ONLINE env var  
5310: 2. Setting HuggingFace offline mode when online access is disabled  
5311: 3. Providing early failure for missing critical dependencies  
5312: 4. Allowing explicit degraded mode marking for optional dependencies  
5313:  
5314: No fallback logic, no "best effort" embeddings. Either dependencies are present  
5315: and configured correctly, or the system fails fast with clear error messages.  
5316: """  
5317:  
5318: import logging  
5319: import os  
5320:
```

```
5321: logger = logging.getLogger(__name__)
5322:
5323:
5324: def _is_model_cached(model_name: str) -> bool:
5325:     """Check if a HuggingFace model is cached locally.
5326:
5327:     Uses a heuristic check of common cache locations to determine if a model
5328:     is likely available offline. This is a best-effort check - false positives
5329:     are acceptable (will fail later when model actually loads), but false negatives
5330:     should be minimized to avoid blocking offline usage of cached models.
5331:
5332:     Args:
5333:         model_name: HuggingFace model name (e.g., "sentence-transformers/model")
5334:
5335:     Returns:
5336:         True if model appears to be cached locally, False otherwise
5337:     """
5338:     from pathlib import Path
5339:
5340:     # Check common HuggingFace cache locations
5341:     cache_dirs = [
5342:         os.path.expanduser("~/cache/huggingface/hub"),
5343:         os.path.expanduser("~/cache/torch/sentence_transformers"),
5344:         os.getenv("HF_HOME"),
5345:         os.getenv("TRANSFORMERS_CACHE"),
5346:     ]
5347:
5348:     # Convert model name to cache directory pattern
5349:     # HF uses "models--org--name" format in cache
5350:     model_slug = model_name.replace("/", "--")
5351:
5352:     for cache_dir in cache_dirs:
5353:         if cache_dir and os.path.exists(cache_dir):
5354:             cache_path = Path(cache_dir)
5355:             # Use glob with specific pattern instead of rglob for efficiency
5356:             # Check just the top-level directories, not recursive
5357:             if any(model_slug in p.name for p in cache_path.iterdir()):
5358:                 return True
5359:
5360:     return False
5361:
5362:
5363: class DependencyLockdownError(RuntimeError):
5364:     """Raised when a dependency constraint is violated."""
5365:     pass
5366:
5367:
5368: class DependencyLockdown:
5369:     """Enforces strict dependency controls to prevent hidden/magic behavior.
5370:
5371:     This class ensures that:
5372:         - Online model downloads are explicitly controlled via HF_ONLINE env var
5373:         - HuggingFace models only download when explicitly allowed
5374:         - Critical dependencies fail fast if missing
5375:         - Optional dependencies are clearly marked as degraded when missing
5376:     """
5377:
```

```
5377:
5378:     def __init__(self) -> None:
5379:         """Initialize dependency lockdown based on environment configuration."""
5380:         self.hf_allowed = os.getenv("HF_ONLINE", "0") == "1"
5381:         self._enforce_offline_mode()
5382:         self._log_configuration()
5383:
5384:     def _enforce_offline_mode(self) -> None:
5385:         """Enforce HuggingFace offline mode if HF_ONLINE is not enabled."""
5386:         if not self.hf_allowed:
5387:             # Set HuggingFace environment variables to prevent downloads
5388:             os.environ["HF_HUB_OFFLINE"] = "1"
5389:             os.environ["TRANSFORMERS_OFFLINE"] = "1"
5390:             logger.info(
5391:                 "Dependency lockdown: HuggingFace offline mode ENFORCED "
5392:                 "(HF_ONLINE=0 or not set)"
5393:             )
5394:         else:
5395:             logger.warning(
5396:                 "Dependency lockdown: HuggingFace online mode ENABLED "
5397:                 "(HF_ONLINE=1). Models may be downloaded from HuggingFace Hub."
5398:             )
5399:
5400:     def _log_configuration(self) -> None:
5401:         """Log current dependency lockdown configuration."""
5402:         logger.info(
5403:             f"Dependency lockdown initialized: "
5404:             f"HF_ONLINE={self.hf_allowed}, "
5405:             f"HF_HUB_OFFLINE={os.getenv('HF_HUB_OFFLINE', 'unset')}, "
5406:             f"TRANSFORMERS_OFFLINE={os.getenv('TRANSFORMERS_OFFLINE', 'unset')}"
5407:         )
5408:
5409:     def check_online_model_access(
5410:         self,
5411:         model_name: str,
5412:         operation: str = "model download"
5413:     ) -> None:
5414:         """Check if online model access is allowed, raise if not.
5415:
5416:         Args:
5417:             model_name: Name of the model being accessed
5418:             operation: Description of the operation (for error message)
5419:
5420:         Raises:
5421:             DependencyLockdownError: If online access is not allowed
5422:         """
5423:         if not self.hf_allowed:
5424:             raise DependencyLockdownError(
5425:                 f"Online model download disabled in this environment. "
5426:                 f"Attempted operation: {operation} for model '{model_name}'. "
5427:                 f"To enable online downloads, set HF_ONLINE=1 environment variable. "
5428:                 f"No fallback to degraded mode - this is a hard failure."
5429:             )
5430:
5431:     def check_critical_dependency(
5432:         self,
```

```
5433:     module_name: str,
5434:     pip_package: str,
5435:     phase: str | None = None
5436: ) -> None:
5437:     """Check if a critical dependency is available, fail fast if not.
5438:
5439:     Args:
5440:         module_name: Python module name to import
5441:         pip_package: pip package name for installation instructions
5442:         phase: Optional phase name where dependency is required
5443:
5444:     Raises:
5445:         DependencyLockdownError: If critical dependency is missing
5446:     """
5447:     try:
5448:         __import__(module_name)
5449:     except ImportError as e:
5450:         phase_info = f" for phase '{phase}'" if phase else ""
5451:         raise DependencyLockdownError(
5452:             f"Critical dependency '{module_name}' is missing{phase_info}. "
5453:             f"Install it with: pip install {pip_package}. "
5454:             f"No degraded mode available - this is a mandatory dependency. "
5455:             f"Original error: {e}"
5456:         ) from e
5457:
5458:     def check_optional_dependency(
5459:         self,
5460:         module_name: str,
5461:         pip_package: str,
5462:         feature: str
5463:     ) -> bool:
5464:         """Check if an optional dependency is available.
5465:
5466:         Args:
5467:             module_name: Python module name to import
5468:             pip_package: pip package name for installation instructions
5469:             feature: Feature name that requires this dependency
5470:
5471:         Returns:
5472:             True if dependency is available, False otherwise
5473:
5474:         Note:
5475:             This does NOT raise an error, but logs a warning about degraded mode.
5476:             Caller must explicitly handle degraded mode and log it clearly.
5477:         """
5478:         try:
5479:             __import__(module_name)
5480:             return True
5481:         except ImportError:
5482:             logger.warning(
5483:                 f"DEGRADED MODE: Optional dependency '{module_name}' not available. "
5484:                 f"Feature '{feature}' will be disabled. "
5485:                 f"Install with: pip install {pip_package}"
5486:             )
5487:             return False
5488:
```

```
5489:     def get_mode_description(self) -> dict[str, str | bool]:
5490:         """Get current dependency lockdown mode description.
5491:
5492:         Returns:
5493:             Dictionary with mode information for logging/debugging
5494:         """
5495:         return {
5496:             "hf_online_allowed": self.hf_allowed,
5497:             "hf_hub_offline": os.getenv("HF_HUB_OFFLINE", "unset"),
5498:             "transformers_offline": os.getenv("TRANSFORMERS_OFFLINE", "unset"),
5499:             "mode": "online" if self.hf_allowed else "offline_enforced",
5500:         }
5501:
5502:
5503: # Global singleton instance
5504: _lockdown_instance: DependencyLockdown | None = None
5505:
5506:
5507: def get_dependency_lockdown() -> DependencyLockdown:
5508:     """Get or create the global dependency lockdown instance.
5509:
5510:     Returns:
5511:         Global DependencyLockdown instance
5512:     """
5513:     global _lockdown_instance
5514:     if _lockdown_instance is None:
5515:         _lockdown_instance = DependencyLockdown()
5516:     return _lockdown_instance
5517:
5518:
5519: def reset_dependency_lockdown() -> None:
5520:     """Reset the global dependency lockdown instance (for testing)."""
5521:     global _lockdown_instance
5522:     _lockdown_instance = None
5523:
5524:
```