

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 5
3: =====
4: Generated: 2025-12-07T06:17:16.261116
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: scripts/pre_commit_validators.py
11: =====
12:
13: #!/usr/bin/env python3
14: """Pre-commit validators for FARFAN calibration system integrity.
15:
16: This module provides validation functions that enforce critical constraints:
17: 1. No hardcoded calibration patterns (weight=, score=, threshold= with numeric literals)
18: 2. JSON schema compliance for all config files
19: 3. SHA256 hash verification against registry
20: 4. YAML prohibition enforcement
21: """
22:
23: import hashlib
24: import json
25: import re
26: import sys
27: from pathlib import Path
28:
29: REPO_ROOT = Path(__file__).resolve().parent.parent
30:
31: CALIBRATION_PATTERNS = [
32:     (r"\bweight\s*=\s*([0-9]+\.[0-9]*)", "weight"),
33:     (r"\bscore\s*=\s*([0-9]+\.[0-9]*)", "score"),
34:     (r"\bthreshold\s*=\s*([0-9]+\.[0-9]*)", "threshold"),
35:     (r"\bmin_evidence\s*=\s*([0-9]+)", "min_evidence"),
36:     (r"\bmax_evidence\s*=\s*([0-9]+)", "max_evidence"),
37:     (r"\bconfidence\s*=\s*([0-9]+\.[0-9]*)", "confidence"),
38:     (r"\bpenalty\s*=\s*([0-9]+\.[0-9]*)", "penalty"),
39:     (r"\btolerance\s*=\s*([0-9]+\.[0-9]*)", "tolerance"),
40:     (r"\bsensitivity\s*=\s*([0-9]+\.[0-9]*)", "sensitivity"),
41:     (r"\bprior\s*=\s*([0-9]+\.[0-9]*)", "prior"),
42: ]
43:
44: ALLOWED_VALUES = {"0", "1", "0.0", "1.0", "0.5", "-1", "2", "10", "100"}
45:
46: EXCLUDED_PATH_SEGMENTS = [
47:     "__pycache__",
48:     ".git",
49:     "farfan-env",
50:     "venv",
51:     ".venv",
52:     "node_modules",
53:     ".eggs",
54:     "dist",
55:     "build",
56: ]
```

```
57:
58: EXCLUDED_PATH_PREFIXES = ["/test/", "/tests/", "config/", "system/config/"]
59:
60: HASH_REGISTRY_FILE = REPO_ROOT / "scripts" / "config_hash_registry.json"
61:
62: JSON_SCHEMA_DEFINITIONS = {
63:     "intrinsic_calibration.json": {
64:         "type": "object",
65:         "properties": {
66:             "_metadata": {"type": "object"},
67:         },
68:         "additionalProperties": {
69:             "type": "object",
70:             "properties": {
71:                 "intrinsic_score": {"type": "array"},
72:                 "b_theory": {"type": "number"},
73:                 "b_impl": {"type": "number"},
74:                 "b_deploy": {"type": "number"},
75:                 "calibration_status": {"type": "string"},
76:                 "layer": {"type": "string"},
77:                 "last_updated": {"type": "string"},
78:             },
79:         },
80:     }
81: }
82:
83:
84: def is_excluded_path(file_path: Path) -> bool:
85:     """Check if file should be excluded from validation."""
86:     path_str = str(file_path)
87:
88:     if "/tmp" in path_str or "/private/var" in path_str:
89:         return False
90:
91:     try:
92:         rel_path = str(file_path.relative_to(REPO_ROOT))
93:     except ValueError:
94:         return False
95:
96:     for segment in EXCLUDED_PATH_SEGMENTS:
97:         if segment in rel_path:
98:             return True
99:
100:     for prefix in EXCLUDED_PATH_PREFIXES:
101:         if rel_path.startswith(prefix) or f"/{prefix}" in rel_path:
102:             return True
103:
104:     return False
105:
106:
107: def is_comment_or_string(line: str) -> bool:
108:     """Check if line is a comment, docstring, or inside string literal."""
109:     stripped = line.strip()
110:
111:     if stripped.startswith("#"):
112:         return True
```

```
113:
114:     if stripped.startswith('""') or stripped.startswith('\'\''):
115:         return True
116:
117:     min_quote_count = 2
118:     return (
119:         stripped.count('\'') >= min_quote_count or stripped.count('\"') >= min_quote_count
120:     )
121:
122:
123: def validate_no_hardcoded_calibrations(
124:     staged_files: list[Path],
125: ) -> tuple[bool, list[str]]:
126:     """Validate that no hardcoded calibration patterns exist in staged Python files.
127:
128:     Args:
129:         staged_files: List of staged file paths
130:
131:     Returns:
132:         Tuple of (is_valid, error_messages)
133:     """
134:     violations = []
135:
136:     for file_path in staged_files:
137:         if not str(file_path).endswith(".py"):
138:             continue
139:
140:         if is_excluded_path(file_path):
141:             continue
142:
143:         try:
144:             content = file_path.read_text(encoding="utf-8")
145:             lines = content.split("\n")
146:
147:             for lineno, line in enumerate(lines, 1):
148:                 if is_comment_or_string(line):
149:                     continue
150:
151:                 for pattern, param_name in CALIBRATION_PATTERNS:
152:                     matches = re.finditer(pattern, line)
153:
154:                     for match in matches:
155:                         value = match.group(1)
156:
157:                         if value not in ALLOWED_VALUES:
158:                             violations.append(
159:                                 f"{file_path}:{lineno}: Hardcoded {param_name}={value}\n"
160:                                 f"  Line: {line.strip()}\n"
161:                                 f"  Calibration values must be loaded from config files."
162:                             )
163:
164:         except (UnicodeDecodeError, OSError) as e:
165:             violations.append(f"Error reading {file_path}: {e}")
166:
167:     if violations:
168:         error_msg = (
```

```
169:         [
170:             "=" * 80,
171:             "COMMIT BLOCKED: Hardcoded calibration values detected",
172:             "=" * 80,
173:             "",
174:             "Found hardcoded calibration patterns:",
175:             "",
176:         ]
177:     + violations
178:     + [
179:         "",
180:         "All calibration parameters must be externalized to:",
181:         "  - config/intrinsic_calibration.json",
182:         "  - system/config/calibration/intrinsic_calibration.json",
183:         "  - src/farfan_pipeline/core/calibration/calibration_registry.py",
184:         "",
185:         "Allowed literal values: 0, 1, 0.0, 1.0, 0.5, -1, 2, 10, 100",
186:         "=" * 80,
187:     ]
188: )
189: return False, error_msg
190:
191: return True, []
192:
193:
194: def validate_json_schema(staged_files: list[Path]) -> tuple[bool, list[str]]:
195:     """Validate JSON files against their schemas.
196:
197:     Args:
198:         staged_files: List of staged file paths
199:
200:     Returns:
201:         Tuple of (is_valid, error_messages)
202:     """
203:     violations = []
204:
205:     for file_path in staged_files:
206:         if not str(file_path).endswith(".json"):
207:             continue
208:
209:         if is_excluded_path(file_path):
210:             continue
211:
212:         try:
213:             with open(file_path, encoding="utf-8") as f:
214:                 data = json.load(f)
215:
216:             file_name = file_path.name
217:             if file_name in JSON_SCHEMA_DEFINITIONS:
218:                 schema = JSON_SCHEMA_DEFINITIONS[file_name]
219:                 if not _validate_against_schema(data, schema):
220:                     violations.append(f"{file_path}: JSON schema validation failed")
221:
222:         except json.JSONDecodeError as e:
223:             violations.append(f"{file_path}: Invalid JSON - {e}")
224:         except Exception as e:
```

```
225:         violations.append(f"{file_path}: Validation error - {e}")
226:
227:     if violations:
228:         error_msg = (
229:             [
230:                 "=" * 80,
231:                 "COMMIT BLOCKED: JSON schema validation failed",
232:                 "=" * 80,
233:                 "",
234:             ]
235:             + violations
236:             + [
237:                 "",
238:                 "Fix JSON schema compliance before committing.",
239:                 "=" * 80,
240:             ]
241:         )
242:         return False, error_msg
243:
244:     return True, []
245:
246:
247: def _validate_against_schema(data: dict, schema: dict) -> bool:
248:     """Basic schema validation (simplified)."""
249:     if schema.get("type") == "object":
250:         if not isinstance(data, dict):
251:             return False
252:
253:         required = schema.get("required", [])
254:         for req_field in required:
255:             if req_field not in data:
256:                 return False
257:
258:     return True
259:
260:
261: def validate_sha256_hashes(staged_files: list[Path]) -> tuple[bool, list[str]]:
262:     """Validate that config file SHA256 hashes match registry.
263:
264:     Args:
265:         staged_files: List of staged file paths
266:
267:     Returns:
268:         Tuple of (is_valid, error_messages)
269:     """
270:     if not HASH_REGISTRY_FILE.exists():
271:         return True, []
272:
273:     try:
274:         with open(HASH_REGISTRY_FILE, encoding="utf-8") as f:
275:             registry = json.load(f)
276:     except Exception as e:
277:         return False, [f"Error loading hash registry: {e}"]
278:
279:     violations = []
280:
```

```
281:     for file_path in staged_files:
282:         if not (str(file_path).endswith(".json") and "config" in str(file_path)):
283:             continue
284:
285:         if is_excluded_path(file_path):
286:             continue
287:
288:         rel_path = str(file_path.relative_to(REPO_ROOT))
289:
290:         if rel_path in registry:
291:             try:
292:                 with open(file_path, "rb") as f:
293:                     actual_hash = hashlib.sha256(f.read()).hexdigest()
294:
295:                     expected_hash = registry[rel_path]["sha256"]
296:
297:                     if actual_hash != expected_hash:
298:                         violations.append(
299:                             f"{file_path}:\n"
300:                             f"  Expected SHA256: {expected_hash}\n"
301:                             f"  Actual SHA256:   {actual_hash}\n"
302:                             f"  Config file modified without updating registry."
303:                         )
304:
305:             except Exception as e:
306:                 violations.append(f"{file_path}: Error computing hash - {e}")
307:
308:     if violations:
309:         error_msg = (
310:             [
311:                 "=" * 80,
312:                 "COMMIT BLOCKED: SHA256 hash mismatch",
313:                 "=" * 80,
314:                 "",
315:                 "Config files modified without updating hash registry:",
316:                 "",
317:             ]
318:             + violations
319:             + [
320:                 "",
321:                 "Update the hash registry with:",
322:                 "  python scripts/update_hash_registry.py",
323:                 "=" * 80,
324:             ]
325:         )
326:         return False, error_msg
327:
328:     return True, []
329:
330:
331: def validate_no_yaml_files(staged_files: list[Path]) -> tuple[bool, list[str]]:
332:     """Validate that no YAML files are being committed.
333:
334:     Args:
335:         staged_files: List of staged file paths
336:
```

```
337: Returns:
338:     Tuple of (is_valid, error_messages)
339:     """
340:     yaml_files = []
341:
342:     for file_path in staged_files:
343:         if is_excluded_path(file_path):
344:             continue
345:
346:         if str(file_path).endswith((".yaml", ".yml")):
347:             yaml_files.append(str(file_path))
348:
349:     if yaml_files:
350:         error_msg = (
351:             [
352:                 "=" * 80,
353:                 "COMMIT BLOCKED: YAML files are prohibited",
354:                 "=" * 80,
355:                 "",
356:                 "Detected YAML files:",
357:                 "",
358:             ]
359:             + [f"  {f}" for f in yaml_files]
360:             + [
361:                 "",
362:                 "FARFAN uses JSON exclusively for configuration.",
363:                 "Convert YAML to JSON before committing.",
364:                 "=" * 80,
365:             ]
366:         )
367:         return False, error_msg
368:
369:     return True, []
370:
371:
372: def main():
373:     """Run all pre-commit validators."""
374:     import subprocess
375:
376:     result = subprocess.run(
377:         ["git", "diff", "--cached", "--name-only", "--diff-filter=ACM"],
378:         capture_output=True,
379:         text=True,
380:         check=False,
381:     )
382:
383:     if result.returncode != 0:
384:         print("Error: Could not get staged files")
385:         return 1
386:
387:     staged_files = [REPO_ROOT / f for f in result.stdout.strip().split("\n") if f]
388:
389:     if not staged_files:
390:         return 0
391:
392:     all_valid = True
```

```
393:     all_errors = []
394:
395:     validators = [
396:         ("Hardcoded Calibrations", validate_no_hardcoded_calibrations),
397:         ("JSON Schema", validate_json_schema),
398:         ("SHA256 Hashes", validate_sha256_hashes),
399:         ("YAML Prohibition", validate_no_yaml_files),
400:     ]
401:
402:     for _name, validator in validators:
403:         is_valid, errors = validator(staged_files)
404:         if not is_valid:
405:             all_valid = False
406:             all_errors.extend(errors)
407:
408:     if not all_valid:
409:         print("\n".join(all_errors), file=sys.stderr)
410:         return 1
411:
412:     print("â\234\223 All pre-commit validations passed")
413:     return 0
414:
415:
416: if __name__ == "__main__":
417:     sys.exit(main())
418:
419:
420:
421: =====
422: FILE: scripts/recommendation_cli.py
423: =====
424:
425: #!/usr/bin/env python3
426: # recommendation_cli.py - CLI for Recommendation Engine
427: # coding=utf-8
428: """
429: Recommendation CLI - Command-line interface for generating recommendations
430:
431: Usage:
432:     python recommendation_cli.py micro --scores scores.json
433:     python recommendation_cli.py meso --clusters clusters.json
434:     python recommendation_cli.py macro --macro-data macro.json
435:     python recommendation_cli.py all --input all_data.json
436:     python recommendation_cli.py demo
437:
438: Examples:
439:     # Generate MICRO recommendations
440:     python recommendation_cli.py micro --scores micro_scores.json -o micro_recs.json
441:
442:     # Generate all recommendations
443:     python recommendation_cli.py all --input sample_data.json -o all_recs.md --format markdown
444:
445:     # Run demonstration
446:     python recommendation_cli.py demo
447: """
448:
```

```
449: import argparse
450: import json
451: import logging
452: import sys
453: from typing import Any
454:
455: from farfan_pipeline.analysis.recommendation_engine import load_recommendation_engine
456:
457: # Configure logging
458: logging.basicConfig(
459:     level=logging.INFO,
460:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
461: )
462: logger = logging.getLogger(__name__)
463:
464: def load_json_file(filepath: str) -> dict[str, Any]:
465:     """Load JSON data from file"""
466:     try:
467:         with open(filepath, encoding='utf-8') as f:
468:             return json.load(f)
469:     except FileNotFoundError:
470:         logger.error(f"File not found: {filepath}")
471:         sys.exit(1)
472:     except json.JSONDecodeError as e:
473:         logger.error(f"Invalid JSON in {filepath}: {e}")
474:         sys.exit(1)
475:
476: def generate_micro(args):
477:     """Generate MICRO-level recommendations"""
478:     logger.info("Generating MICRO-level recommendations...")
479:
480:     # Load scores
481:     scores = load_json_file(args.scores)
482:
483:     # Load engine
484:     engine = load_recommendation_engine(args.rules, args.schema)
485:
486:     # Generate recommendations
487:     rec_set = engine.generate_micro_recommendations(scores)
488:
489:     # Output
490:     logger.info(f"Generated {rec_set.rules_matched} recommendations from {rec_set.total_rules_evaluated} rules")
491:
492:     if args.output:
493:         engine.export_recommendations(
494:             {'MICRO': rec_set},
495:             args.output,
496:             format=args.format
497:         )
498:         logger.info(f"Saved to {args.output}")
499:     else:
500:         # Print to stdout
501:         if args.format == 'json':
502:             print(json.dumps(rec_set.to_dict(), indent=2, ensure_ascii=False))
503:         else:
504:             print(engine._format_as_markdown({'MICRO': rec_set}))
```

```
505:
506: def generate_meso(args):
507:     """Generate MESO-level recommendations"""
508:     logger.info("Generating MESO-level recommendations...")
509:
510:     # Load cluster data
511:     cluster_data = load_json_file(args.clusters)
512:
513:     # Load engine
514:     engine = load_recommendation_engine(args.rules, args.schema)
515:
516:     # Generate recommendations
517:     rec_set = engine.generate_meso_recommendations(cluster_data)
518:
519:     # Output
520:     logger.info(f"Generated {rec_set.rules_matched} recommendations from {rec_set.total_rules_evaluated} rules")
521:
522:     if args.output:
523:         engine.export_recommendations(
524:             {'MESO': rec_set},
525:             args.output,
526:             format=args.format
527:         )
528:         logger.info(f"Saved to {args.output}")
529:     else:
530:         # Print to stdout
531:         if args.format == 'json':
532:             print(json.dumps(rec_set.to_dict(), indent=2, ensure_ascii=False))
533:         else:
534:             print(engine._format_as_markdown({'MESO': rec_set}))
535:
536: def generate_macro(args):
537:     """Generate MACRO-level recommendations"""
538:     logger.info("Generating MACRO-level recommendations...")
539:
540:     # Load macro data
541:     macro_data = load_json_file(args.macro_data)
542:
543:     # Load engine
544:     engine = load_recommendation_engine(args.rules, args.schema)
545:
546:     # Generate recommendations
547:     rec_set = engine.generate_macro_recommendations(macro_data)
548:
549:     # Output
550:     logger.info(f"Generated {rec_set.rules_matched} recommendations from {rec_set.total_rules_evaluated} rules")
551:
552:     if args.output:
553:         engine.export_recommendations(
554:             {'MACRO': rec_set},
555:             args.output,
556:             format=args.format
557:         )
558:         logger.info(f"Saved to {args.output}")
559:     else:
560:         # Print to stdout
```

```
561:         if args.format == 'json':
562:             print(json.dumps(rec_set.to_dict(), indent=2, ensure_ascii=False))
563:         else:
564:             print(engine._format_as_markdown({'MACRO': rec_set}))
565:
566: def generate_all(args):
567:     """Generate recommendations at all levels"""
568:     logger.info("Generating recommendations at all levels...")
569:
570:     # Load input data
571:     data = load_json_file(args.input)
572:
573:     micro_scores = data.get('micro_scores', {})
574:     cluster_data = data.get('cluster_data', {})
575:     macro_data = data.get('macro_data', {})
576:
577:     # Load engine
578:     engine = load_recommendation_engine(args.rules, args.schema)
579:
580:     # Generate all recommendations
581:     all_recs = engine.generate_all_recommendations(
582:         micro_scores, cluster_data, macro_data
583:     )
584:
585:     # Output summary
586:     logger.info(f"MICRO: {all_recs['MICRO'].rules_matched} recommendations")
587:     logger.info(f"MESO: {all_recs['MESO'].rules_matched} recommendations")
588:     logger.info(f"MACRO: {all_recs['MACRO'].rules_matched} recommendations")
589:
590:     if args.output:
591:         engine.export_recommendations(all_recs, args.output, format=args.format)
592:         logger.info(f"Saved to {args.output}")
593:     else:
594:         # Print to stdout
595:         if args.format == 'json':
596:             output = {level: rec_set.to_dict() for level, rec_set in all_recs.items()}
597:             print(json.dumps(output, indent=2, ensure_ascii=False))
598:         else:
599:             print(engine._format_as_markdown(all_recs))
600:
601: def run_demo(args):
602:     """Run demonstration with sample data"""
603:     logger.info("Running demonstration...")
604:
605:     # Sample MICRO scores
606:     micro_scores = {
607:         'PA01-DIM01': 1.2, # Below threshold
608:         'PA02-DIM02': 1.5, # Below threshold
609:         'PA03-DIM05': 1.4, # Below threshold
610:         'PA04-DIM03': 2.0, # Above threshold
611:     }
612:
613:     # Sample cluster data
614:     cluster_data = {
615:         'CL01': {
616:             'score': 72.0,
```

```
617:         'variance': 0.25,
618:         'weak_pa': 'PA02'
619:     },
620:     'CL02': {
621:         'score': 58.0,
622:         'variance': 0.12,
623:     },
624:     'CL03': {
625:         'score': 65.0,
626:         'variance': 0.28,
627:         'weak_pa': 'PA04'
628:     }
629: }
630:
631: # Sample macro data
632: macro_data = {
633:     'macro_band': 'SATISFACTORIO',
634:     'clusters_below_target': ['CL02', 'CL03'],
635:     'variance_alert': 'MODERADA',
636:     'priority_micro_gaps': ['PA01-DIM05', 'PA05-DIM04', 'PA04-DIM04', 'PA08-DIM05']
637: }
638:
639: # Load engine
640: engine = load_recommendation_engine(args.rules, args.schema)
641:
642: # Generate all recommendations
643: all_recs = engine.generate_all_recommendations(
644:     micro_scores, cluster_data, macro_data
645: )
646:
647: # Display results
648: print("\n" + "=" * 80)
649: print("DEMONSTRATION: Recommendation Engine")
650: print("=" * 80)
651:
652: print("\nð\237\223\212 INPUT DATA:")
653: print(f"  MICRO Scores: {len(micro_scores)} PA-DIM combinations")
654: print(f"  MESO Clusters: {len(cluster_data)} clusters")
655: print(f"  MACRO Band: {macro_data['macro_band']}")
656:
657: print("\nð\237\223\213 RESULTS:")
658: print(f"  MICRO: {all_recs['MICRO'].rules_matched} recommendations (from {all_recs['MICRO'].total_rules_evaluated} rules)")
659: print(f"  MESO:  {all_recs['MESO'].rules_matched} recommendations (from {all_recs['MESO'].total_rules_evaluated} rules)")
660: print(f"  MACRO: {all_recs['MACRO'].rules_matched} recommendations (from {all_recs['MACRO'].total_rules_evaluated} rules)")
661:
662: # Show sample MICRO recommendations
663: if all_recs['MICRO'].recommendations:
664:     print("\n" + "-" * 80)
665:     print("SAMPLE MICRO RECOMMENDATION:")
666:     print("-" * 80)
667:     rec = all_recs['MICRO'].recommendations[0]
668:     print(f"Rule ID: {rec.rule_id}")
669:     print(f"Problem: {rec.problem[:200]}...")
670:     print(f"Intervention: {rec.intervention[:200]}...")
671:     print(f"Responsible: {rec.responsible['entity']}")
672:     print(f"Horizon: {rec.horizon['start']} â\206\222 {rec.horizon['end']}")
```

```
673:
674:     # Show sample MESO recommendations
675:     if all_recs['MESO'].recommendations:
676:         print("\n" + "-" * 80)
677:         print("SAMPLE MESO RECOMMENDATION:")
678:         print("-" * 80)
679:         rec = all_recs['MESO'].recommendations[0]
680:         print(f"Rule ID: {rec.rule_id}")
681:         print(f"Cluster: {rec.metadata.get('cluster_id')}")
682:         print(f"Score: {rec.metadata.get('score'):.1f} ({rec.metadata.get('score_band')})")
683:         print(f"Intervention: {rec.intervention[:200]}...")
684:
685:     # Show sample MACRO recommendations
686:     if all_recs['MACRO'].recommendations:
687:         print("\n" + "-" * 80)
688:         print("SAMPLE MACRO RECOMMENDATION:")
689:         print("-" * 80)
690:         rec = all_recs['MACRO'].recommendations[0]
691:         print(f"Rule ID: {rec.rule_id}")
692:         print(f"Band: {rec.metadata.get('macro_band')}")
693:         print(f"Intervention: {rec.intervention[:200]}...")
694:
695:     print("\n" + "=" * 80)
696:
697:     # Optionally save
698:     if args.output:
699:         engine.export_recommendations(all_recs, args.output, format=args.format)
700:         logger.info(f"Full report saved to {args.output}")
701:
702: def main():
703:     """Main CLI entry point"""
704:     parser = argparse.ArgumentParser(
705:         description='Generate rule-based recommendations for policy plans',
706:         formatter_class=argparse.RawDescriptionHelpFormatter,
707:         epilog=__doc__
708:     )
709:
710:     # Global arguments
711:     parser.add_argument(
712:         '--rules',
713:         default='config/recommendation_rules.json',
714:         help='Path to recommendation rules JSON file'
715:     )
716:     parser.add_argument(
717:         '--schema',
718:         default='rules/recommendation_rules.schema.json',
719:         help='Path to rules schema JSON file'
720:     )
721:
722:     # Subcommands
723:     subparsers = parser.add_subparsers(dest='command', help='Command to execute')
724:
725:     # MICRO command
726:     micro_parser = subparsers.add_parser('micro', help='Generate MICRO-level recommendations')
727:     micro_parser.add_argument('--scores', required=True, help='Path to scores JSON file')
728:     micro_parser.add_argument('-o', '--output', help='Output file path')
```

```

729:     micro_parser.add_argument('--format', choices=['json', 'markdown'], default='json', help='Output format')
730:     micro_parser.set_defaults(func=generate_micro)
731:
732:     # MESO command
733:     meso_parser = subparsers.add_parser('meso', help='Generate MESO-level recommendations')
734:     meso_parser.add_argument('--clusters', required=True, help='Path to cluster data JSON file')
735:     meso_parser.add_argument('-o', '--output', help='Output file path')
736:     meso_parser.add_argument('--format', choices=['json', 'markdown'], default='json', help='Output format')
737:     meso_parser.set_defaults(func=generate_meso)
738:
739:     # MACRO command
740:     macro_parser = subparsers.add_parser('macro', help='Generate MACRO-level recommendations')
741:     macro_parser.add_argument('--macro-data', required=True, help='Path to macro data JSON file')
742:     macro_parser.add_argument('-o', '--output', help='Output file path')
743:     macro_parser.add_argument('--format', choices=['json', 'markdown'], default='json', help='Output format')
744:     macro_parser.set_defaults(func=generate_macro)
745:
746:     # ALL command
747:     all_parser = subparsers.add_parser('all', help='Generate recommendations at all levels')
748:     all_parser.add_argument('--input', required=True, help='Path to combined input JSON file')
749:     all_parser.add_argument('-o', '--output', help='Output file path')
750:     all_parser.add_argument('--format', choices=['json', 'markdown'], default='json', help='Output format')
751:     all_parser.set_defaults(func=generate_all)
752:
753:     # DEMO command
754:     demo_parser = subparsers.add_parser('demo', help='Run demonstration with sample data')
755:     demo_parser.add_argument('-o', '--output', help='Output file path')
756:     demo_parser.add_argument('--format', choices=['json', 'markdown'], default='markdown', help='Output format')
757:     demo_parser.set_defaults(func=run_demo)
758:
759:     # Parse arguments
760:     args = parser.parse_args()
761:
762:     if not args.command:
763:         parser.print_help()
764:         sys.exit(1)
765:
766:     # Execute command
767:     args.func(args)
768:
769: if __name__ == '__main__':
770:     main()
771:
772:
773:
774: =====
775: FILE: scripts/refactor_parameter_loader.py
776: =====
777:
778: #!/usr/bin/env python3
779: """Script to refactor all get_parameter_loader() calls to ParameterLoaderV2.get()."""
780:
781: import re
782: import sys
783: from pathlib import Path
784: from typing import List, Tuple

```

```
785:
786:
787: def refactor_file(file_path: Path) -> Tuple[int, List[str]]:
788:     """
789:     Refactor a single file to use ParameterLoaderV2.
790:
791:     Returns:
792:     Tuple of (replacement_count, issues_list)
793:     """
794:     with open(file_path, 'r', encoding='utf-8') as f:
795:         content = f.read()
796:
797:     original_content = content
798:     issues = []
799:     replacements = 0
800:
801:     # Pattern 1: get_parameter_loader().get("method_id").get("param_name", default)
802:     pattern1 = r'get_parameter_loader\(\)\.get\(["(\\[^\"]+)"\]\.get\(["(\\[^\"]+)"(?:,\\s*\\([^\"]+))\\?)'\
803:     matches1 = list(re.finditer(pattern1, content))
804:
805:     for match in reversed(matches1):
806:         method_id = match.group(1)
807:         param_name = match.group(2)
808:         default = match.group(3) if match.group(3) else "None"
809:
810:         replacement = f'ParameterLoaderV2.get("{method_id}", "{param_name}", {default})'
811:         content = content[:match.start()] + replacement + content[match.end():]
812:         replacements += 1
813:
814:     # Pattern 2: _PARAM_LOADER = get_parameter_loader() - remove these global assignments
815:     pattern2 = r'^\\s*_PARAM_LOADER\\s*=\\s*get_parameter_loader\\(\\)\\s*$'
816:     if re.search(pattern2, content, re.MULTILINE):
817:         content = re.sub(pattern2, '', content, flags=re.MULTILINE)
818:         replacements += 1
819:
820:     # Pattern 3: from farfan_pipeline import get_parameter_loader
821:     pattern3 = r'from\\s+farfan_pipeline\\s+import\\s+get_parameter_loader\\s*\\n'
822:     if re.search(pattern3, content):
823:         # Replace with ParameterLoaderV2 import
824:         content = re.sub(
825:             pattern3,
826:             'from farfan_pipeline.core.parameters import ParameterLoaderV2\\n',
827:             content
828:         )
829:         replacements += 1
830:
831:     # Check if file still has get_parameter_loader references (except in definition)
832:     remaining = re.findall(r'get_parameter_loader\\(\\)', content)
833:     if remaining and '__init__.py' not in str(file_path):
834:         issues.append(f"Still has {len(remaining)} get_parameter_loader() calls")
835:
836:     # Only write if changed
837:     if content != original_content:
838:         with open(file_path, 'w', encoding='utf-8') as f:
839:             f.write(content)
840:
```

```
841:     return replacements, issues
842:
843:
844: def main():
845:     """Main refactoring execution."""
846:     src_path = Path("src")
847:
848:     if not src_path.exists():
849:         print("ERROR: src/ directory not found")
850:         sys.exit(1)
851:
852:     # Find all Python files
853:     py_files = list(src_path.rglob("*.py"))
854:
855:     # Exclude __init__.py that defines get_parameter_loader
856:     exclude_files = [
857:         "src/farfan_pipeline/__init__.py", # Defines get_parameter_loader
858:     ]
859:
860:     total_replacements = 0
861:     total_issues = []
862:     files_modified = 0
863:
864:     for py_file in py_files:
865:         if str(py_file) in exclude_files:
866:             continue
867:
868:         try:
869:             replacements, issues = refactor_file(py_file)
870:             if replacements > 0:
871:                 files_modified += 1
872:                 total_replacements += replacements
873:                 print(f"â\234\223 {py_file}: {replacements} replacements")
874:
875:             if issues:
876:                 total_issues.extend([f"{py_file}: {issue}" for issue in issues])
877:                 print(f"  â\232 ï,\217  {' '.join(issues)}")
878:
879:         except Exception as e:
880:             print(f"ERROR processing {py_file}: {e}")
881:             total_issues.append(f"{py_file}: {str(e)}")
882:
883:     print(f"\n{'='*70}")
884:     print(f"SUMMARY:")
885:     print(f"  Files modified: {files_modified}")
886:     print(f"  Total replacements: {total_replacements}")
887:     print(f"  Issues: {len(total_issues)}")
888:
889:     if total_issues:
890:         print(f"\nISSUES:")
891:         for issue in total_issues[:10]:
892:             print(f"  - {issue}")
893:         if len(total_issues) > 10:
894:             print(f"  ... and {len(total_issues) - 10} more")
895:
896:     return 0 if len(total_issues) == 0 else 1
```

```
897:
898:
899: if __name__ == "__main__":
900:     sys.exit(main())
901:
902:
903:
904: =====
905: FILE: scripts/run_mutation_suite.py
906: =====
907:
908: #!/usr/bin/env python3
909: """
910: Advanced Verification Suite (Mutation & Coverage)
911: Runs mutmut and coverage as requested by governance policy.
912: """
913: import os
914: import sys
915: import subprocess
916:
917: def run_command(cmd, description):
918:     print(f"\n--- {description} ---")
919:     print(f"Command: {cmd}")
920:     try:
921:         env = os.environ.copy()
922:         cwd = os.getcwd()
923:         farfan_core_path = os.path.join(cwd, "farfan_core")
924:         env["PYTHONPATH"] = f"{farfan_core_path}:{env.get('PYTHONPATH', '')}"
925:
926:         subprocess.check_call(cmd, shell=True, env=env)
927:         print("\234\205 PASSED")
928:         return True
929:     except subprocess.CalledProcessError as e:
930:         print(f"\235\214 FAILED (Exit Code: {e.returncode})")
931:         return False
932:
933: def main():
934:     print("=== ADVANCED VERIFICATION SUITE ===")
935:
936:     # 1. Coverage Analysis
937:     print("\n[1/3] Running Coverage Analysis...")
938:     cov_cmd = (
939:         "python -m pytest farfan_core/farfan_core/contracts/tests -v "
940:         "--cov=farfan_core/farfan_core/contracts "
941:         "--cov-report=term-missing "
942:         "--cov-report=xml "
943:         "--cov-fail-under=90"
944:     )
945:     if not run_command(cov_cmd, "Coverage Check (90% threshold)"):
946:         print("\232 i,\217 Coverage below 90% threshold")
947:         return False
948:
949:     # 2. Mutation Testing
950:     print("\n[2/3] Running Mutation Testing (mutmut)...")
951:     # Initialize mutmut if not already done
952:     init_cmd = "mutmut run --paths-to-mutate=farfan_core/farfan_core/contracts --tests-dir=farfan_core/farfan_core/contracts/tests || true"
```

```
953:     run_command(init_cmd, "Mutation Testing")
954:
955:     # Generate mutation report
956:     report_cmd = "mutmut show all > mutation_report.txt 2>&1 || true"
957:     run_command(report_cmd, "Mutation Report Generation")
958:
959:     # 3. Determinism Check
960:     print("\n[3/3] Verifying Determinism...")
961:     det_cmd = "python -m pytest farfan_core/contracts/tests/test_cdc.py -v"
962:     if not run_command(det_cmd, "Determinism Matrix"):
963:         return False
964:
965:     print("\n=== MUTATION SUITE COMPLETE ===")
966:     return True
967:
968: if __name__ == "__main__":
969:     success = main()
970:     sys.exit(0 if success else 1)
971:
972:
973:
974: =====
975: FILE: scripts/test_parameter_loader_v2.py
976: =====
977:
978: #!/usr/bin/env python3
979: """Test script for ParameterLoaderV2."""
980:
981: import sys
982: from pathlib import Path
983:
984: def test_imports():
985:     """Test that all imports work."""
986:     print("Testing imports...")
987:     try:
988:         from farfan_pipeline.core.parameters import ParameterLoaderV2
989:         print(" â\234\223 ParameterLoaderV2 import successful")
990:     except ImportError as e:
991:         print(f" â\234\227 ParameterLoaderV2 import failed: {e}")
992:         return False
993:
994:     try:
995:         from farfan_pipeline.core.calibration.calibration_registry import get_calibration
996:         print(" â\234\223 get_calibration import successful")
997:     except ImportError as e:
998:         print(f" â\234\227 get_calibration import failed: {e}")
999:         return False
1000:
1001:     try:
1002:         from farfan_pipeline.core.calibration.decorators import calibrated_method
1003:         print(" â\234\223 calibrated_method import successful")
1004:     except ImportError as e:
1005:         print(f" â\234\227 calibrated_method import failed: {e}")
1006:         return False
1007:
1008:     return True
```

```
1009:
1010:
1011: def test_parameter_loading():
1012:     """Test parameter loading functionality."""
1013:     print("\nTesting parameter loading...")
1014:     from farfan_pipeline.core.parameters import ParameterLoaderV2
1015:
1016:     # Test 1: Load specific parameter
1017:     value = ParameterLoaderV2.get(
1018:         "farfan_core.analysis.derek_beach.BayesianThresholdsConfig",
1019:         "kl_divergence",
1020:         0.0
1021:     )
1022:     if value == 0.01:
1023:         print(f" â\234\223 Single parameter load: kl_divergence={value}")
1024:     else:
1025:         print(f" â\234\227 Single parameter load failed: expected 0.01, got {value}")
1026:         return False
1027:
1028:     # Test 2: Load all parameters
1029:     all_params = ParameterLoaderV2.get_all(
1030:         "farfan_core.analysis.derek_beach.BayesianThresholdsConfig"
1031:     )
1032:     if len(all_params) == 5:
1033:         print(f" â\234\223 All parameters load: {len(all_params)} params")
1034:     else:
1035:         print(f" â\234\227 All parameters load failed: expected 5, got {len(all_params)}")
1036:         return False
1037:
1038:     # Test 3: get_calibration compatibility
1039:     from farfan_pipeline.core.calibration.calibration_registry import get_calibration
1040:     compat_params = get_calibration(
1041:         "farfan_core.analysis.derek_beach.BayesianThresholdsConfig"
1042:     )
1043:     if len(compat_params) == 5:
1044:         print(f" â\234\223 Backward compatibility: get_calibration works")
1045:     else:
1046:         print(f" â\234\227 Backward compatibility failed")
1047:         return False
1048:
1049:     return True
1050:
1051:
1052: def test_nonexistent_method():
1053:     """Test handling of nonexistent methods."""
1054:     print("\nTesting nonexistent method handling...")
1055:     from farfan_pipeline.core.parameters import ParameterLoaderV2
1056:
1057:     value = ParameterLoaderV2.get("nonexistent.method", "param", "default_value")
1058:     if value == "default_value":
1059:         print(" â\234\223 Default value returned for nonexistent method")
1060:         return True
1061:     else:
1062:         print(f" â\234\227 Expected 'default_value', got {value}")
1063:         return False
1064:
```

```
1065:
1066: def main():
1067:     """Run all tests."""
1068:     print("="*70)
1069:     print("PARAMETER LOADER V2 FUNCTIONALITY TEST")
1070:     print("="*70)
1071:
1072:     tests = [
1073:         ("Imports", test_imports),
1074:         ("Parameter Loading", test_parameter_loading),
1075:         ("Nonexistent Method", test_nonexistent_method),
1076:     ]
1077:
1078:     results = []
1079:     for name, test_func in tests:
1080:         try:
1081:             passed = test_func()
1082:             results.append((name, passed))
1083:         except Exception as e:
1084:             print(f"â\234\227 {name} raised exception: {e}")
1085:             results.append((name, False))
1086:
1087:     print("\n" + "="*70)
1088:     print("TEST SUMMARY")
1089:     print("="*70)
1090:
1091:     all_passed = all(passed for _, passed in results)
1092:     for name, passed in results:
1093:         status = "â\234\223 PASS" if passed else "â\234\227 FAIL"
1094:         print(f" {status}: {name}")
1095:
1096:     if all_passed:
1097:         print("\nâ\234\223 ALL TESTS PASSED")
1098:         return 0
1099:     else:
1100:         print("\nâ\234\227 SOME TESTS FAILED")
1101:         return 1
1102:
1103:
1104: if __name__ == "__main__":
1105:     sys.exit(main())
1106:
1107:
1108:
1109: =====
1110: FILE: scripts/update_hash_registry.py
1111: =====
1112:
1113: #!/usr/bin/env python3
1114: """Update SHA256 hash registry for config files.
1115:
1116: This script scans config directories and updates the hash registry
1117: with SHA256 checksums for all JSON configuration files.
1118: """
1119:
1120: import hashlib
```

```
1121: import json
1122: from datetime import datetime
1123: from pathlib import Path
1124:
1125: REPO_ROOT = Path(__file__).resolve().parent.parent
1126:
1127: CONFIG_DIRECTORIES = [
1128:     "config",
1129:     "system/config",
1130: ]
1131:
1132: HASH_REGISTRY_FILE = REPO_ROOT / "scripts" / "config_hash_registry.json"
1133:
1134:
1135: def compute_sha256(file_path: Path) -> str:
1136:     """Compute SHA256 hash of file."""
1137:     sha256_hash = hashlib.sha256()
1138:     with open(file_path, "rb") as f:
1139:         for byte_block in iter(lambda: f.read(4096), b''):
1140:             sha256_hash.update(byte_block)
1141:     return sha256_hash.hexdigest()
1142:
1143:
1144: def scan_config_files() -> dict[str, dict]:
1145:     """Scan config directories and compute hashes."""
1146:     registry = {}
1147:
1148:     for config_dir in CONFIG_DIRECTORIES:
1149:         config_path = REPO_ROOT / config_dir
1150:
1151:         if not config_path.exists():
1152:             print(f"Warning: {config_dir} does not exist")
1153:             continue
1154:
1155:         for json_file in config_path.rglob("*.json"):
1156:             if "__pycache__" in str(json_file):
1157:                 continue
1158:
1159:             rel_path = str(json_file.relative_to(REPO_ROOT))
1160:
1161:             try:
1162:                 sha256 = compute_sha256(json_file)
1163:                 file_size = json_file.stat().st_size
1164:
1165:                 registry[rel_path] = {
1166:                     "sha256": sha256,
1167:                     "size_bytes": file_size,
1168:                     "last_updated": datetime.utcnow().isoformat() + "Z",
1169:                 }
1170:
1171:                 print(f"â234\223 {rel_path}: {sha256[:16]}...")
1172:
1173:             except Exception as e:
1174:                 print(f"â234\227 {rel_path}: Error - {e}")
1175:
1176:     return registry
```

```
1177:
1178:
1179: def main():
1180:     """Update hash registry."""
1181:     print("Scanning config files...")
1182:     print()
1183:
1184:     registry = scan_config_files()
1185:
1186:     print()
1187:     print(f"Found {len(registry)} config files")
1188:
1189:     HASH_REGISTRY_FILE.parent.mkdir(parents=True, exist_ok=True)
1190:
1191:     with open(HASH_REGISTRY_FILE, "w", encoding="utf-8") as f:
1192:         json.dump(registry, f, indent=2, sort_keys=True)
1193:
1194:     print(f"\nHash registry updated: {HASH_REGISTRY_FILE}")
1195:     print("Commit this file with your config changes.")
1196:
1197:
1198: if __name__ == "__main__":
1199:     main()
1200:
1201:
1202:
1203: =====
1204: FILE: scripts/validate_monolith.py
1205: =====
1206:
1207: #!/usr/bin/env python3
1208: """
1209: Validate a questionnaire monolith against the JSON Schema and basic semantic rules.
1210:
1211: Exit code:
1212: - 0 if schema validation and semantic checks pass.
1213: - 1 otherwise.
1214:
1215: Outputs validation_report.json with:
1216: {
1217:     "validation_passed": bool,
1218:     "errors": [...],
1219:     "warnings": [...],
1220:     "schema_hash": "sha256:<hash>"
1221: }
1222: """
1223:
1224: from __future__ import annotations
1225:
1226: import argparse
1227: import hashlib
1228: import json
1229: import sys
1230: from pathlib import Path
1231: from typing import Any
1232:
```

```
1233: from jsonschema import Draft7Validator
1234:
1235:
1236: ROOT = Path(__file__).resolve().parent.parent
1237: DEFAULT_MONOLITH = ROOT / "system" / "config" / "questionnaire" / "questionnaire_monolith.json"
1238: DEFAULT_SCHEMA = ROOT / "system" / "config" / "questionnaire" / "questionnaire_schema.json"
1239: DEFAULT_REGISTRY = ROOT / "system" / "config" / "questionnaire" / "pattern_registry.json"
1240: DEFAULT_REPORT = ROOT / "validation_report.json"
1241: DEFAULT_DEPRECATION = ROOT / "system" / "config" / "questionnaire" / "pattern_deprecation.json"
1242:
1243:
1244: def load_json(path: Path) -> Any:
1245:     with path.open("r", encoding="utf-8") as f:
1246:         return json.load(f)
1247:
1248:
1249: def compute_sha256(path: Path) -> str:
1250:     data = path.read_bytes()
1251:     digest = hashlib.sha256(data).hexdigest()
1252:     return f"sha256:{digest}"
1253:
1254:
1255: def semantic_checks(monolith: dict[str, Any]) -> list[dict[str, Any]]:
1256:     """Lightweight semantic checks beyond JSON Schema."""
1257:     errors: list[dict[str, Any]] = []
1258:
1259:     required_keys = ["canonical_notation", "blocks", "schema_version", "integrity"]
1260:     missing = [k for k in required_keys if k not in monolith]
1261:     if missing:
1262:         errors.append({"type": "semantic", "message": f"Missing required keys: {missing}"})
1263:     return errors
1264:
1265:     blocks = monolith.get("blocks", {})
1266:     required_blocks = [
1267:         "macro_question",
1268:         "meso_questions",
1269:         "micro_questions",
1270:         "niveles_abstraccion",
1271:         "scoring",
1272:         "semantic_layers",
1273:     ]
1274:     missing_blocks = [b for b in required_blocks if b not in blocks]
1275:     if missing_blocks:
1276:         errors.append({"type": "semantic", "message": f"Missing required blocks: {missing_blocks}"})
1277:
1278:     micro_questions = blocks.get("micro_questions", [])
1279:     if not isinstance(micro_questions, list) or not micro_questions:
1280:         errors.append({"type": "semantic", "message": "micro_questions must be a non-empty list"})
1281:     return errors
1282:
1283:     first_q = micro_questions[0]
1284:     required_q_fields = ["question_id", "text", "cluster_id", "dimension_id"]
1285:     missing_q = [f for f in required_q_fields if f not in first_q]
1286:     if missing_q:
1287:         errors.append(
1288:             {"type": "semantic", "message": f"Micro questions missing required fields: {missing_q}"}
```

```
1289:     )
1290:
1291:     return errors
1292:
1293:
1294: def validate_pattern_refs(
1295:     monolith: dict[str, Any], registry_path: Path, deprecation_path: Path
1296: ) -> tuple[list[dict[str, Any]], list[dict[str, Any]]]:
1297:     """Validate that all pattern_ref entries resolve to the registry and find unused/deprecated patterns."""
1298:     errors: list[dict[str, Any]] = []
1299:     warnings: list[dict[str, Any]] = []
1300:
1301:     if not registry_path.exists():
1302:         warnings.append(
1303:             {
1304:                 "type": "registry",
1305:                 "message": f"Registry file not found at {registry_path}",
1306:             }
1307:         )
1308:     return errors, warnings
1309:
1310: registry = load_json(registry_path)
1311: deprecated: set[str] = set()
1312: if deprecation_path.exists():
1313:     dep = load_json(deprecation_path)
1314:     deprecated = set(dep.get("deprecated_patterns", []))
1315: registry_index = {entry.get("pattern_id"): entry for entry in registry}
1316: used: set[str] = set()
1317:
1318: for mq in monolith.get("blocks", {}).get("micro_questions", []):
1319:     for pattern in mq.get("patterns", []) or []:
1320:         ref = pattern.get("pattern_ref")
1321:         if ref:
1322:             if ref not in registry_index:
1323:                 errors.append(
1324:                     {
1325:                         "type": "registry",
1326:                         "message": f"pattern_ref '{ref}' not found in registry",
1327:                         "instance_path": ["blocks", "micro_questions", mq.get("question_id"), "patterns"],
1328:                     }
1329:                 )
1330:             else:
1331:                 used.add(ref)
1332:
1333: unused = sorted(set(registry_index) - used)
1334: if unused:
1335:     warnings.append(
1336:         {
1337:             "type": "registry",
1338:             "message": f"Unused registry patterns: {len(unused)}",
1339:             "details": unused,
1340:         }
1341:     )
1342:
1343: deprecated_in_use = sorted(ref for ref in used if ref in deprecated)
1344: if deprecated_in_use:
```

```
1345:         warnings.append(
1346:             {
1347:                 "type": "registry",
1348:                 "message": f"Deprecated patterns referenced: {len(deprecated_in_use)}",
1349:                 "details": deprecated_in_use,
1350:             }
1351:         )
1352:
1353:     return errors, warnings
1354:
1355:
1356: def validate(
1357:     monolith_path: Path, schema_path: Path, registry_path: Path, deprecation_path: Path
1358: ) -> dict[str, Any]:
1359:     monolith = load_json(monolith_path)
1360:     schema = load_json(schema_path)
1361:
1362:     validator = Draft7Validator(schema)
1363:     schema_errors = sorted(validator.iter_errors(monolith), key=lambda e: e.path)
1364:
1365:     errors: list[dict[str, Any]] = []
1366:     for err in schema_errors:
1367:         errors.append(
1368:             {
1369:                 "type": "schema",
1370:                 "message": err.message,
1371:                 "instance_path": list(err.path),
1372:                 "schema_path": list(err.schema_path),
1373:             }
1374:         )
1375:
1376:     errors.extend(semantic_checks(monolith))
1377:     registry_errors, registry_warnings = validate_pattern_refs(
1378:         monolith, registry_path, deprecation_path
1379:     )
1380:     errors.extend(registry_errors)
1381:
1382:     validation_passed = len(errors) == 0
1383:     report = {
1384:         "validation_passed": validation_passed,
1385:         "errors": errors,
1386:         "warnings": registry_warnings,
1387:         "schema_hash": compute_sha256(schema_path),
1388:     }
1389:     return report
1390:
1391:
1392: def main() -> int:
1393:     parser = argparse.ArgumentParser(description="Validate questionnaire monolith.")
1394:     parser.add_argument(
1395:         "--monolith",
1396:         type=Path,
1397:         default=DEFAULT_MONOLITH,
1398:         help=f"Path to questionnaire monolith (default: {DEFAULT_MONOLITH})",
1399:     )
1400:     parser.add_argument(
```

```
1401:     "--schema",
1402:     type=Path,
1403:     default=DEFAULT_SCHEMA,
1404:     help=f"Path to JSON Schema (default: {DEFAULT_SCHEMA})",
1405: )
1406: parser.add_argument(
1407:     "--registry",
1408:     type=Path,
1409:     default=DEFAULT_REGISTRY,
1410:     help=f"Path to pattern registry (default: {DEFAULT_REGISTRY})",
1411: )
1412: parser.add_argument(
1413:     "--deprecation",
1414:     type=Path,
1415:     default=DEFAULT_DEPRECATION,
1416:     help=f"Path to deprecated patterns list (default: {DEFAULT_DEPRECATION})",
1417: )
1418: parser.add_argument(
1419:     "--report",
1420:     type=Path,
1421:     default=DEFAULT_REPORT,
1422:     help=f"Path to write validation report (default: {DEFAULT_REPORT})",
1423: )
1424: args = parser.parse_args()
1425:
1426: report = validate(args.monolith, args.schema, args.registry, args.deprecation)
1427: args.report.write_text(json.dumps(report, ensure_ascii=False, indent=2), encoding="utf-8")
1428:
1429: if report["validation_passed"]:
1430:     return 0
1431:
1432: return 1
1433:
1434:
1435: if __name__ == "__main__":
1436:     sys.exit(main())
1437:
1438:
1439:
1440: =====
1441: FILE: scripts/validate_phase2_architecture.py
1442: =====
1443:
1444: #!/usr/bin/env python3
1445: """
1446: Comprehensive Phase 2 Validation Suite
1447:
1448: Validates that all 12 jobfront requirements are met:
1449: 1. SPC â222 PreprocessedDocument (60 chunks, metadata)
1450: 2. Questionnaire structure (300 micro, 4 meso, 1 macro)
1451: 3. ChunkRouter & routing table
1452: 4. 30 Executors registered
1453: 5. MethodRegistry coverage
1454: 6. Signal Registry completeness
1455: 7-12. Runtime components (tested separately)
1456:
```

```
1457: Exit code 0 = ALL PASS
1458: Exit code 1 = AT LEAST ONE FAILURE
1459: ""
1460:
1461: import json
1462: import sys
1463: from pathlib import Path
1464: from typing import Any
1465:
1466: # Color codes for terminal output
1467: GREEN = "\033[92m"
1468: RED = "\033[91m"
1469: YELLOW = "\033[93m"
1470: RESET = "\033[0m"
1471: BOLD = "\033[1m"
1472:
1473: class ValidationReport:
1474:     """Tracks validation results across all jobfronts."""
1475:
1476:     def __init__(self):
1477:         self.results = []
1478:         self.failures = []
1479:
1480:     def add_pass(self, jobfront: str, check: str, details: str = ""):
1481:         self.results.append({
1482:             "jobfront": jobfront,
1483:             "check": check,
1484:             "status": "PASS",
1485:             "details": details
1486:         })
1487:         print(f"{GREEN}\033[234\223{RESET} [{jobfront}] {check}")
1488:         if details:
1489:             print(f"    {details}")
1490:
1491:     def add_fail(self, jobfront: str, check: str, reason: str):
1492:         self.results.append({
1493:             "jobfront": jobfront,
1494:             "check": check,
1495:             "status": "FAIL",
1496:             "reason": reason
1497:         })
1498:         self.failures.append(f"{jobfront}: {check}")
1499:         print(f"{RED}\033[234\227{RESET} [{jobfront}] {check}")
1500:         print(f"    {RED}REASON: {reason}{RESET}")
1501:
1502:     def print_summary(self):
1503:         total = len(self.results)
1504:         passed = sum(1 for r in self.results if r["status"] == "PASS")
1505:         failed = len(self.failures)
1506:
1507:         print("\n" + "="*80)
1508:         print(f"{BOLD}VALIDATION SUMMARY{RESET}")
1509:         print("="*80)
1510:         print(f"Total checks: {total}")
1511:         print(f"{GREEN}Passed: {passed}{RESET}")
1512:         print(f"{RED}Failed: {failed}{RESET}")
```

```
1513:
1514:     if self.failures:
1515:         print(f"\n{RED}{BOLD}FAILED CHECKS:{RESET}")
1516:         for failure in self.failures:
1517:             print(f" - {failure}")
1518:     else:
1519:         print(f"\n{GREEN}{BOLD}ð\237\216\211 ALL VALIDATION CHECKS PASSED!{RESET}")
1520:
1521:     return failed == 0
1522:
1523:
1524: def validate_jobfront_2_questionnaire(report: ValidationReport):
1525:     """Validate questionnaire_monolith.json structure."""
1526:     jobfront = "JF02-Questionnaire"
1527:
1528:     monolith_path = Path("system/config/questionnaire/questionnaire_monolith.json")
1529:
1530:     if not monolith_path.exists():
1531:         report.add_fail(jobfront, "File exists", f"{monolith_path} not found")
1532:         return
1533:
1534:     report.add_pass(jobfront, "File exists", str(monolith_path))
1535:
1536:     try:
1537:         with open(monolith_path, "r", encoding="utf-8") as f:
1538:             monolith = json.load(f)
1539:     except Exception as e:
1540:         report.add_fail(jobfront, "JSON parse", str(e))
1541:         return
1542:
1543:     report.add_pass(jobfront, "JSON parse", "Valid JSON")
1544:
1545:     # Check structure
1546:     if "blocks" not in monolith:
1547:         report.add_fail(jobfront, "Structure", "Missing 'blocks' key")
1548:         return
1549:
1550:     blocks = monolith["blocks"]
1551:
1552:     # Micro questions
1553:     micro = blocks.get("micro_questions", [])
1554:     if len(micro) == 300:
1555:         report.add_pass(jobfront, "300 micro_questions", f"Found {len(micro)}")
1556:     else:
1557:         report.add_fail(jobfront, "300 micro_questions", f"Expected 300, found {len(micro)}")
1558:
1559:     # Meso questions
1560:     meso = blocks.get("meso_questions", [])
1561:     if len(meso) == 4:
1562:         report.add_pass(jobfront, "4 meso_questions", f"Found {len(meso)}")
1563:     else:
1564:         report.add_fail(jobfront, "4 meso_questions", f"Expected 4, found {len(meso)}")
1565:
1566:     # Macro question
1567:     macro = blocks.get("macro_question")
1568:     if macro:
```

```
1569:     report.add_pass(jobfront, "1 macro_question", "Found")
1570: else:
1571:     report.add_fail(jobfront, "1 macro_question", "Missing")
1572:
1573: # Validate micro_question fields
1574: if micro:
1575:     sample = micro[0]
1576:     required_fields = [
1577:         "question_id", "question_global", "base_slot",
1578:         "dimension_id", "policy_area_id", "cluster_id",
1579:         "scoring_modality", "expected_elements", "method_sets"
1580:     ]
1581:
1582:     missing = [f for f in required_fields if f not in sample]
1583:     if not missing:
1584:         report.add_pass(jobfront, "Micro fields complete", f"All {len(required_fields)} required fields present")
1585:     else:
1586:         report.add_fail(jobfront, "Micro fields complete", f"Missing: {missing}")
1587:
1588:     # Check unique question_global
1589:     globals_set = {q.get("question_global") for q in micro if "question_global" in q}
1590:     if len(globals_set) == len(micro):
1591:         report.add_pass(jobfront, "question_global unique", "All 300 are unique")
1592:     else:
1593:         report.add_fail(jobfront, "question_global unique", f"Duplicates found: {len(micro) - len(globals_set)}")
1594:
1595:
1596: def validate_jobfront_3_chunk_router(report: ValidationReport):
1597:     """Validate ChunkRouter existence and routing table."""
1598:     jobfront = "JF03-ChunkRouter"
1599:
1600:     router_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator/chunk_router.py")
1601:
1602:     if not router_path.exists():
1603:         report.add_fail(jobfront, "File exists", f"{router_path} not found")
1604:         return
1605:
1606:     report.add_pass(jobfront, "File exists", str(router_path))
1607:
1608:     content = router_path.read_text()
1609:
1610:     # Check for ChunkRouter class
1611:     if "class ChunkRouter:" in content:
1612:         report.add_pass(jobfront, "ChunkRouter class", "Found")
1613:     else:
1614:         report.add_fail(jobfront, "ChunkRouter class", "Not found")
1615:
1616:     # Check for ROUTING_TABLE
1617:     if "ROUTING_TABLE" in content and "chunk_type" in content:
1618:         report.add_pass(jobfront, "ROUTING_TABLE", "Found")
1619:     else:
1620:         report.add_fail(jobfront, "ROUTING_TABLE", "Not found")
1621:
1622:     # Check for version
1623:     if "ROUTING_TABLE_VERSION" in content:
1624:         report.add_pass(jobfront, "ROUTING_TABLE_VERSION", "Immutable version defined")
```

```
1625:     else:
1626:         report.add_fail(jobfront, "ROUTING_TABLE_VERSION", "Version not found")
1627:
1628:
1629: def validate_jobfront_4_executors(report: ValidationReport):
1630:     """Validate 30 executor registration."""
1631:     jobfront = "JF04-Executors"
1632:
1633:     exec_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator/executors.py")
1634:
1635:     if not exec_path.exists():
1636:         report.add_fail(jobfront, "File exists", f"{exec_path} not found")
1637:         return
1638:
1639:     report.add_pass(jobfront, "File exists", str(exec_path))
1640:
1641:     content = exec_path.read_text()
1642:
1643:     # Check EXECUTOR_REGISTRY
1644:     if "EXECUTOR_REGISTRY" in content:
1645:         report.add_pass(jobfront, "EXECUTOR_REGISTRY", "Found")
1646:     else:
1647:         report.add_fail(jobfront, "EXECUTOR_REGISTRY", "Not found")
1648:         return
1649:
1650:     # Count registrations - look for D{n}-Q{m} patterns
1651:     expected_slots = [
1652:         f"D{d}-Q{q}" for d in range(1, 7) for q in range(1, 6)
1653:     ]
1654:
1655:     found_slots = [slot for slot in expected_slots if f'"{slot}"' in content]
1656:
1657:     if len(found_slots) == 30:
1658:         report.add_pass(jobfront, "30 executors registered", f"All {len(found_slots)} base_slots found")
1659:     else:
1660:         missing = set(expected_slots) - set(found_slots)
1661:         report.add_fail(jobfront, "30 executors registered", f"Missing {len(missing)}: {missing}")
1662:
1663:
1664: def validate_jobfront_5_method_registry(report: ValidationReport):
1665:     """Validate MethodRegistry exists."""
1666:     jobfront = "JF05-MethodRegistry"
1667:
1668:     registry_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator/method_registry.py")
1669:
1670:     if not registry_path.exists():
1671:         report.add_fail(jobfront, "File exists", f"{registry_path} not found")
1672:         return
1673:
1674:     report.add_pass(jobfront, "File exists", str(registry_path))
1675:
1676:     content = registry_path.read_text()
1677:
1678:     if "class MethodRegistry:" in content:
1679:         report.add_pass(jobfront, "MethodRegistry class", "Found")
1680:     else:
```

```
1681:         report.add_fail(jobfront, "MethodRegistry class", "Not found")
1682:
1683:     # Check for lazy instantiation
1684:     if "get_method" in content and "_instance_cache" in content:
1685:         report.add_pass(jobfront, "Lazy instantiation", "get_method + cache found")
1686:     else:
1687:         report.add_fail(jobfront, "Lazy instantiation", "Missing implementation")
1688:
1689:
1690: def validate_jobfront_6_signal_registry(report: ValidationReport):
1691:     """Validate signal registry components."""
1692:     jobfront = "JF06-SignalRegistry"
1693:
1694:     signal_files = [
1695:         "signal_registry.py",
1696:         "signal_aliasing.py",
1697:         "signal_cache_invalidation.py"
1698:     ]
1699:
1700:     base_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator")
1701:
1702:     for filename in signal_files:
1703:         file_path = base_path / filename
1704:         if file_path.exists():
1705:             report.add_pass(jobfront, f"{filename} exists", str(file_path))
1706:         else:
1707:             report.add_fail(jobfront, f"{filename} exists", f"{file_path} not found")
1708:
1709:
1710: def validate_jobfront_9_evidence_model(report: ValidationReport):
1711:     """Validate Evidence components."""
1712:     jobfront = "JF09-Evidence"
1713:
1714:     evidence_files = {
1715:         "evidence_assembler.py": "EvidenceAssembler",
1716:         "evidence_validator.py": "EvidenceValidator",
1717:         "evidence_registry.py": "EvidenceRegistry"
1718:     }
1719:
1720:     base_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator")
1721:
1722:     for filename, class_name in evidence_files.items():
1723:         file_path = base_path / filename
1724:         if file_path.exists():
1725:             content = file_path.read_text()
1726:             if f"class {class_name}" in content:
1727:                 report.add_pass(jobfront, f"{class_name}", f"Found in {filename}")
1728:             else:
1729:                 report.add_fail(jobfront, f"{class_name}", f"Class not found in {filename}")
1730:         else:
1731:             report.add_fail(jobfront, f"{filename} exists", f"{file_path} not found")
1732:
1733:
1734: def validate_jobfront_11_seed_registry(report: ValidationReport):
1735:     """Validate seed registry for determinism."""
1736:     jobfront = "JF11-SeedRegistry"
```

```
1737:
1738:     seed_path = Path("farfan_pipeline/farfan_pipeline/core/orchestrator/seed_registry.py")
1739:
1740:     if seed_path.exists():
1741:         report.add_pass(jobfront, "seed_registry.py exists", str(seed_path))
1742:     else:
1743:         report.add_fail(jobfront, "seed_registry.py exists", f"{seed_path} not found")
1744:
1745:
1746: def main():
1747:     """Run all validation checks."""
1748:     print(f"\n{BOLD}{'='*80}{RESET}")
1749:     print(f"{BOLD}Phase 2 - Comprehensive Validation Suite{RESET}")
1750:     print(f"{BOLD}{'='*80}{RESET}\n")
1751:
1752:     report = ValidationReport()
1753:
1754:     # Run validations
1755:     validate_jobfront_2_questionnaire(report)
1756:     validate_jobfront_3_chunk_router(report)
1757:     validate_jobfront_4_executors(report)
1758:     validate_jobfront_5_method_registry(report)
1759:     validate_jobfront_6_signal_registry(report)
1760:     validate_jobfront_9_evidence_model(report)
1761:     validate_jobfront_11_seed_registry(report)
1762:
1763:     # Summary
1764:     all_passed = report.print_summary()
1765:
1766:     # Save report
1767:     report_path = Path("validation_report.json")
1768:     with open(report_path, "w", encoding="utf-8") as f:
1769:         json.dump({
1770:             "results": report.results,
1771:             "summary": {
1772:                 "total": len(report.results),
1773:                 "passed": sum(1 for r in report.results if r["status"] == "PASS"),
1774:                 "failed": len(report.failures)
1775:             }
1776:         }, f, indent=2)
1777:
1778:     print(f"\n{BOLD}Report saved to: {report_path}{RESET}")
1779:
1780:     sys.exit(0 if all_passed else 1)
1781:
1782:
1783: if __name__ == "__main__":
1784:     main()
1785:
1786:
1787:
1788: =====
1789: FILE: scripts/validate_workflows.py
1790: =====
1791:
1792: #!/usr/bin/env python3
```

```
1793: """
1794: Validate GitHub Actions workflow YAML files
1795: """
1796: import sys
1797: import yaml
1798: from pathlib import Path
1799:
1800: def validate_workflow(filepath):
1801:     """Validate a single workflow YAML file."""
1802:     try:
1803:         with open(filepath, 'r') as f:
1804:             data = yaml.safe_load(f)
1805:
1806:             # Check required fields
1807:             if 'name' not in data:
1808:                 print(f"\235\214 {filepath.name}: Missing 'name' field")
1809:                 return False
1810:
1811:             if 'on' not in data:
1812:                 print(f"\235\214 {filepath.name}: Missing 'on' field")
1813:                 return False
1814:
1815:             if 'jobs' not in data:
1816:                 print(f"\235\214 {filepath.name}: Missing 'jobs' field")
1817:                 return False
1818:
1819:             print(f"\234\205 {filepath.name}: Valid workflow")
1820:             print(f"    Name: {data['name']}")
1821:             print(f"    Jobs: {' , '.join(data['jobs'].keys())}")
1822:             return True
1823:
1824:     except yaml.YAMLError as e:
1825:         print(f"\235\214 {filepath.name}: YAML syntax error")
1826:         print(f"    {e}")
1827:         return False
1828:
1829:     except Exception as e:
1830:         print(f"\235\214 {filepath.name}: Error - {e}")
1831:         return False
1832:
1833: def main():
1834:     workflows_dir = Path('.github/workflows')
1835:
1836:     if not workflows_dir.exists():
1837:         print("\235\214 .github/workflows directory not found")
1838:         sys.exit(1)
1839:
1840:     print("=== Validating GitHub Actions Workflows ===\n")
1841:
1842:     workflow_files = list(workflows_dir.glob('*.*.yaml')) + list(workflows_dir.glob('*.*.yml'))
1843:
1844:     if not workflow_files:
1845:         print("\235\214 No workflow files found")
1846:         sys.exit(1)
1847:
1848:     results = []
1849:     for workflow_file in sorted(workflow_files):
```

```
1849:         result = validate_workflow(workflow_file)
1850:         results.append(result)
1851:         print()
1852:
1853:     print("=" * 50)
1854:     passed = sum(results)
1855:     total = len(results)
1856:     print(f"Results: {passed}/{total} workflows valid")
1857:
1858:     if passed == total:
1859:         print("â\234\205 All workflows are valid!")
1860:         sys.exit(0)
1861:     else:
1862:         print("â\235\214 Some workflows have errors")
1863:         sys.exit(1)
1864:
1865: if __name__ == '__main__':
1866:     main()
1867:
1868:
1869:
1870: =====
1871: FILE: scripts/validators/validate_no_relative_imports.py
1872: =====
1873:
1874: #!/usr/bin/env python3
1875: """Validator to ensure no relative imports exist in farfan_pipeline.
1876:
1877: This script enforces the absolute-imports-only policy by scanning all Python
1878: files in src/farfan_pipeline/ and failing if any relative imports are found.
1879:
1880: Exit codes:
1881:     0: No relative imports found (success)
1882:     1: Relative imports found (failure)
1883:     2: Script error (e.g., directory not found)
1884:
1885: Usage:
1886:     python scripts/validators/validate_no_relative_imports.py
1887: """
1888:
1889: import ast
1890: import sys
1891: from pathlib import Path
1892: from typing import List, Tuple
1893:
1894:
1895: def find_relative_imports(file_path: Path) -> List[Tuple[int, str]]:
1896:     """Find all relative imports in a Python file.
1897:
1898:     Args:
1899:         file_path: Path to Python file
1900:
1901:     Returns:
1902:         List of (line_number, import_statement) tuples for relative imports
1903:     """
1904:     try:
```

```
1905:     content = file_path.read_text(encoding="utf-8")
1906:     tree = ast.parse(content, filename=str(file_path))
1907: except (SyntaxError, UnicodeDecodeError) as e:
1908:     print(f"â\232 ï\217 Could not parse {file_path}: {e}", file=sys.stderr)
1909:     return []
1910:
1911: violations = []
1912:
1913: for node in ast.walk(tree):
1914:     if isinstance(node, (ast.ImportFrom,)):
1915:         # ImportFrom with level > 0 indicates relative import
1916:         if hasattr(node, 'level') and node.level and node.level > 0:
1917:             line_text = content.split('\n')[node.lineno - 1].strip()
1918:             violations.append((node.lineno, line_text))
1919:
1920: return violations
1921:
1922:
1923: def main() -> int:
1924:     """Scan farfan_pipeline for relative imports and report violations."""
1925:     repo_root = Path(__file__).resolve().parent.parent
1926:     src_dir = repo_root / "src" / "farfan_pipeline"
1927:
1928:     if not src_dir.exists():
1929:         print(f"â\235\214 Source directory not found: {src_dir}", file=sys.stderr)
1930:         return 2
1931:
1932:     print("ð\237\224\215 Scanning for relative imports in farfan_pipeline...")
1933:     print(f"    Root: {src_dir}\n")
1934:
1935:     all_violations = []
1936:
1937:     for py_file in src_dir.rglob("*.py"):
1938:         violations = find_relative_imports(py_file)
1939:         if violations:
1940:             all_violations.append((py_file, violations))
1941:
1942:     if not all_violations:
1943:         print("â\234\205 No relative imports found. All imports are absolute.")
1944:         return 0
1945:
1946:     print(f"â\235\214 Found relative imports in {len(all_violations)} file(s):\n")
1947:
1948:     for file_path, violations in all_violations:
1949:         rel_path = file_path.relative_to(repo_root)
1950:         print(f"    {rel_path}:")
1951:         for line_num, line_text in violations:
1952:             print(f"        Line {line_num}: {line_text}")
1953:         print()
1954:
1955:     print("Policy violation: All imports must be absolute.")
1956:     print("Example: Use 'from farfan_pipeline.core import x' instead of 'from . import x'")
1957:
1958:     return 1
1959:
1960:
```

```
1961: if __name__ == "__main__":
1962:     sys.exit(main())
1963:
1964:
1965:
1966: =====
1967: FILE: scripts/validators/verify_no_scattered_loaders.py
1968: =====
1969:
1970: #!/usr/bin/env python3
1971: """
1972: Verification script: Ensures complete eradication of scattered parameter loader calls.
1973:
1974: FAILURE CONDITIONS:
1975: 1. Any get_parameter_loader() call outside src/core/parameters/
1976: 2. Any CALIBRATIONS = { dict found anywhere in codebase
1977: 3. Script exits with error code and descriptive message
1978: """
1979:
1980: import re
1981: import subprocess
1982: import sys
1983: from pathlib import Path
1984: from typing import List, Tuple
1985:
1986:
1987: def run_grep(pattern: str, path: str = "src/") -> List[str]:
1988:     """Run grep and return matching lines."""
1989:     try:
1990:         result = subprocess.run(
1991:             ["grep", "-rn", pattern, path, "--include=*.py"],
1992:             capture_output=True,
1993:             text=True,
1994:             check=False
1995:         )
1996:         if result.returncode == 0:
1997:             return [line.strip() for line in result.stdout.split('\n') if line.strip()]
1998:         return []
1999:     except Exception as e:
2000:         print(f"WARNING: grep failed: {e}")
2001:         return []
2002:
2003:
2004: def check_scattered_loaders() -> Tuple[bool, List[str]]:
2005:     """
2006:     Check for scattered get_parameter_loader() calls.
2007:
2008:     Returns:
2009:         (success, violations) where success=True if no violations found
2010:     """
2011:     violations = []
2012:
2013:     # Find all get_parameter_loader() calls
2014:     loader_calls = run_grep(r"get_parameter_loader\(\\)")
2015:
2016:     # Filter out allowed locations
```

```
2017:     allowed_patterns = [
2018:         r"src/farfan_pipeline/___init___\.py.*def get_parameter_loader",
2019:         r"src/farfan_pipeline/___init___\.py.*get_parameter_loader\\(\)",
2020:         r"src/farfan_pipeline/___init___\.py.*___all__.*get_parameter_loader",
2021:         r"src/farfan_pipeline/core/calibration/.*",
2022:         r"src/farfan_pipeline/core/parameters/.*",
2023:         r"#.*get_parameter_loader", # Comments
2024:         r"___all__.*get_parameter_loader", # ___all__ exports
2025:     ]
2026:
2027:     for call in loader_calls:
2028:         is_allowed = False
2029:         for pattern in allowed_patterns:
2030:             if re.search(pattern, call):
2031:                 is_allowed = True
2032:                 break
2033:
2034:         if not is_allowed:
2035:             violations.append(f"SCATTERED LOADER: {call}")
2036:
2037:     return len(violations) == 0, violations
2038:
2039:
2040: def check_calibrations_dict() -> Tuple[bool, List[str]]:
2041:     """
2042:     Check for CALIBRATIONS = { dict definitions.
2043:
2044:     Returns:
2045:         (success, violations) where success=True if no violations found
2046:     """
2047:     violations = []
2048:
2049:     # Find CALIBRATIONS dict definitions
2050:     calibrations_defs = run_grep(r"CALIBRATIONS\s*=\s*\{")
2051:
2052:     # No CALIBRATIONS dict should exist anywhere
2053:     for definition in calibrations_defs:
2054:         violations.append(f"CALIBRATIONS DICT: {definition}")
2055:
2056:     return len(violations) == 0, violations
2057:
2058:
2059: def check_old_imports() -> Tuple[bool, List[str]]:
2060:     """
2061:     Check for old parameter_loader imports (not from core.parameters).
2062:
2063:     Returns:
2064:         (success, violations) where success=True if no violations found
2065:     """
2066:     violations = []
2067:
2068:     # Find imports
2069:     import_lines = run_grep(r"from.*parameter_loader import")
2070:
2071:     # Filter out new approved imports
2072:     approved_patterns = [
```

```
2073:         r"from farfan_pipeline\.core\.parameters import",
2074:         r"from \.core\.parameters import",
2075:         r"from farfan_pipeline\.core\.calibration\.parameter_loader import ParameterLoader",
2076:         r"from \.core\.calibration\.parameter_loader import ParameterLoader",
2077:     ]
2078:
2079:     for import_line in import_lines:
2080:         is_approved = False
2081:         for pattern in approved_patterns:
2082:             if re.search(pattern, import_line):
2083:                 is_approved = True
2084:                 break
2085:
2086:         if not is_approved:
2087:             violations.append(f"OLD IMPORT: {import_line}")
2088:
2089:     return len(violations) == 0, violations
2090:
2091:
2092: def main() -> int:
2093:     """
2094:     Main verification execution.
2095:
2096:     Returns:
2097:         0 if all checks pass, 1 if any failures
2098:     """
2099:     print("="*70)
2100:     print("PARAMETER LOADER ERADICATION VERIFICATION")
2101:     print("="*70)
2102:
2103:     all_passed = True
2104:     all_violations = []
2105:
2106:     # Check 1: Scattered loaders
2107:     print("\n[1/3] Checking for scattered get_parameter_loader() calls...")
2108:     passed, violations = check_scattered_loaders()
2109:     if passed:
2110:         print("    â234\223 PASS: No scattered loader calls found")
2111:     else:
2112:         print(f"    â234\227 FAIL: Found {len(violations)} scattered loader calls")
2113:         all_passed = False
2114:         all_violations.extend(violations)
2115:
2116:     # Check 2: CALIBRATIONS dict
2117:     print("\n[2/3] Checking for CALIBRATIONS = { dict definitions...}")
2118:     passed, violations = check_calibrations_dict()
2119:     if passed:
2120:         print("    â234\223 PASS: No CALIBRATIONS dict found")
2121:     else:
2122:         print(f"    â234\227 FAIL: Found {len(violations)} CALIBRATIONS dict definitions")
2123:         all_passed = False
2124:         all_violations.extend(violations)
2125:
2126:     # Check 3: Old imports
2127:     print("\n[3/3] Checking for old parameter_loader imports...")
2128:     passed, violations = check_old_imports()
```

```
2129:     if passed:
2130:         print("  â\234\223 PASS: No problematic imports found")
2131:     else:
2132:         print(f"  â\234\227 FAIL: Found {len(violations)} problematic imports")
2133:         all_passed = False
2134:         all_violations.extend(violations)
2135:
2136:     # Final verdict
2137:     print("\n" + "="*70)
2138:     if all_passed:
2139:         print("â\234\223 VERIFICATION PASSED")
2140:         print("  All scattered parameter loaders have been eradicated.")
2141:         print("  Centralized ParameterLoaderV2 is the single source of truth.")
2142:         return 0
2143:     else:
2144:         print("â\234\227 VERIFICATION FAILED")
2145:         print(f"  loader eradication incomplete - {len(all_violations)} violations remain")
2146:         print("\nVIOLATIONS:")
2147:         for violation in all_violations[:20]:
2148:             print(f"    - {violation}")
2149:         if len(all_violations) > 20:
2150:             print(f"    ... and {len(all_violations) - 20} more")
2151:         return 1
2152:
2153:
2154: if __name__ == "__main__":
2155:     sys.exit(main())
2156:
2157:
2158:
2159: =====
2160: FILE: scripts/validators/verify_parameter_consistency.py
2161: =====
2162:
2163: #!/usr/bin/env python3
2164: """
2165: Parameter Consistency Verification Script
2166:
2167: VERIFICATION CONDITIONS:
2168: 1. Every parameter has all required fields
2169: 2. No parameter has both required=true AND has_default=true
2170: 3. default_type is one of [literal, expression, complex]
2171: 4. Determinism: re-run produces identical results
2172:
2173: FAILURE CONDITION:
2174: - If ANY parameter violates invariant or if re-run produces different results,
2175:   ABORT with 'parameter extraction inconsistent' error
2176: - No parametrizable method may lack input_parameters block
2177: """
2178: import hashlib
2179: import json
2180: import sys
2181: from pathlib import Path
2182: from typing import Any
2183:
2184:
```

```
2185: class ParameterConsistencyError(Exception):
2186:     """Raised when parameter consistency checks fail."""
2187:     pass
2188:
2189:
2190: def compute_hash(data: dict[str, Any]) -> str:
2191:     """Compute deterministic hash of JSON data."""
2192:     json_str = json.dumps(data, sort_keys=True, indent=None)
2193:     return hashlib.sha256(json_str.encode('utf-8')).hexdigest()
2194:
2195:
2196: def validate_parameter(param: dict[str, Any], method_id: str) -> list[str]:
2197:     """
2198:     Validate a single parameter descriptor.
2199:
2200:     Returns list of error messages (empty if valid).
2201:     """
2202:     errors = []
2203:
2204:     required_fields = ['name', 'type_hint', 'has_default', 'required',
2205:                        'default_value', 'default_type', 'default_source']
2206:
2207:     for field in required_fields:
2208:         if field not in param:
2209:             errors.append(
2210:                 f"Parameter '{param.get('name', 'UNKNOWN')}' in method '{method_id}' "
2211:                 f"missing required field: {field}"
2212:             )
2213:
2214:     if len(errors) > 0:
2215:         return errors
2216:
2217:     if param['required'] and param['has_default']:
2218:         errors.append(
2219:             f"Parameter '{param['name']}' in method '{method_id}' "
2220:             f"violates invariant: required=True AND has_default=True"
2221:         )
2222:
2223:     if param['required'] != (not param['has_default']):
2224:         errors.append(
2225:             f"Parameter '{param['name']}' in method '{method_id}' "
2226:             f"violates invariant: required != !has_default"
2227:         )
2228:
2229:     if param['has_default'] and param['default_type'] is not None:
2230:         valid_types = ['literal', 'expression', 'complex']
2231:         if param['default_type'] not in valid_types:
2232:             errors.append(
2233:                 f"Parameter '{param['name']}' in method '{method_id}' "
2234:                 f"has invalid default_type: '{param['default_type']}' "
2235:                 f"(must be one of {valid_types})"
2236:             )
2237:
2238:     return errors
2239:
2240:
```

```
2241: def validate_method(method_id: str, method_data: dict[str, Any]) -> list[str]:
2242:     """
2243:     Validate a single method's parameter structure.
2244:
2245:     Returns list of error messages (empty if valid).
2246:     """
2247:     errors = []
2248:
2249:     signature = method_data.get('signature', {})
2250:     requiere_parametrizacion = signature.get('requiere_parametrizacion', False)
2251:     input_parameters = signature.get('input_parameters')
2252:
2253:     if requiere_parametrizacion:
2254:         if input_parameters is None:
2255:             errors.append(
2256:                 f"Method '{method_id}' has requiere_parametrizacion=true "
2257:                 f"but lacks input_parameters block"
2258:             )
2259:             return errors
2260:
2261:         if not isinstance(input_parameters, list):
2262:             errors.append(
2263:                 f"Method '{method_id}' input_parameters is not a list"
2264:             )
2265:             return errors
2266:
2267:         for param in input_parameters:
2268:             param_errors = validate_parameter(param, method_id)
2269:             errors.extend(param_errors)
2270:
2271:     return errors
2272:
2273: def validate_inventory(inventory_path: Path) -> tuple[list[str], str]:
2274:     """
2275:     Validate inventory file for parameter consistency.
2276:
2277:     Returns (errors, hash) tuple.
2278:     """
2279:
2280:     if not inventory_path.exists():
2281:         raise FileNotFoundError(f"Inventory file not found: {inventory_path}")
2282:
2283:     with open(inventory_path) as f:
2284:         data = json.load(f)
2285:
2286:     inventory_hash = compute_hash(data)
2287:
2288:     errors = []
2289:     methods = data.get('methods', {})
2290:
2291:     for method_id, method_data in methods.items():
2292:         method_errors = validate_method(method_id, method_data)
2293:         errors.extend(method_errors)
2294:
2295:     return errors, inventory_hash
2296:
```

```
2297:
2298: def run_extraction() -> Path:
2299:     """
2300:     Run the method inventory extraction.
2301:
2302:     Returns path to generated inventory file.
2303:     """
2304:     output_path = Path("artifacts/test_runs/method_inventory_verification.json")
2305:     output_path.parent.mkdir(parents=True, exist_ok=True)
2306:
2307:     try:
2308:         from farfan_pipeline.core.method_inventory import (
2309:             build_method_inventory,
2310:             method_inventory_to_json,
2311:         )
2312:
2313:         inventory = build_method_inventory()
2314:         data = method_inventory_to_json(inventory)
2315:
2316:         with open(output_path, 'w') as f:
2317:             json.dump(data, f, indent=2)
2318:
2319:         return output_path
2320:     except Exception as e:
2321:         raise RuntimeError(f"Extraction failed: {e}") from e
2322:
2323:
2324: def main() -> int:
2325:     """Main verification routine."""
2326:     print("=" * 80)
2327:     print("PARAMETER CONSISTENCY VERIFICATION")
2328:     print("=" * 80)
2329:
2330:     try:
2331:         print("\n[1/4] Running first extraction...")
2332:         inventory_path_1 = run_extraction()
2333:         print(f"â234\223 Generated: {inventory_path_1}")
2334:
2335:         print("\n[2/4] Validating first extraction...")
2336:         errors_1, hash_1 = validate_inventory(inventory_path_1)
2337:
2338:         if errors_1:
2339:             print(f"â234\227 FAILED: {len(errors_1)} errors found")
2340:             for error in errors_1[:10]:
2341:                 print(f"â234\227 - {error}")
2342:             if len(errors_1) > 10:
2343:                 print(f"â234\227 ... and {len(errors_1) - 10} more errors")
2344:             raise ParameterConsistencyError(
2345:                 f"parameter extraction inconsistent: {len(errors_1)} validation errors"
2346:             )
2347:
2348:         print(f"â234\223 All parameters valid")
2349:         print(f"â234\223 Hash: {hash_1[:16]}...")
2350:
2351:         print("\n[3/4] Running second extraction (determinism check)...")
2352:         inventory_path_2 = run_extraction()
```

```
2353:     print(f"        â\234\223 Generated: {inventory_path_2}")
2354:
2355:     print("\n[4/4] Comparing hashes...")
2356:     errors_2, hash_2 = validate_inventory(inventory_path_2)
2357:
2358:     if errors_2:
2359:         print(f"        â\234\227 FAILED: Second run has {len(errors_2)} errors")
2360:         raise ParameterConsistencyError(
2361:             "parameter extraction inconsistent: second run validation failed"
2362:         )
2363:
2364:     if hash_1 != hash_2:
2365:         print("        â\234\227 FAILED: Hashes differ")
2366:         print(f"        Run 1: {hash_1}")
2367:         print(f"        Run 2: {hash_2}")
2368:         raise ParameterConsistencyError(
2369:             "parameter extraction inconsistent: non-deterministic results"
2370:         )
2371:
2372:     print(f"        â\234\223 Hashes match: {hash_2[:16]}...")
2373:
2374:     print("\n" + "=" * 80)
2375:     print("â\234\223 VERIFICATION PASSED")
2376:     print("=" * 80)
2377:     print(" - All parameters have required fields")
2378:     print(" - No parameter violates required/has_default invariant")
2379:     print(" - All default_types are valid")
2380:     print(" - Extraction is deterministic")
2381:     print("=" * 80)
2382:
2383:     return 0
2384:
2385: except ParameterConsistencyError as e:
2386:     print("\n" + "=" * 80)
2387:     print("â\234\227 VERIFICATION FAILED")
2388:     print("=" * 80)
2389:     print(f"ERROR: {e}")
2390:     print("=" * 80)
2391:     return 1
2392:
2393: except Exception as e:
2394:     print("\n" + "=" * 80)
2395:     print("â\234\227 VERIFICATION ERROR")
2396:     print("=" * 80)
2397:     print(f"Unexpected error: {e}")
2398:     import traceback
2399:     traceback.print_exc()
2400:     print("=" * 80)
2401:     return 2
2402:
2403:
2404: if __name__ == "__main__":
2405:     sys.exit(main())
2406:
2407:
2408:
```

```
2409: =====
2410: FILE: scripts/verify_layer_config.py
2411: =====
2412:
2413: #!/usr/bin/env python3
2414: import json
2415: from pathlib import Path
2416:
2417: config_file = Path(__file__).parent.parent / "config" / "canonic_inventorry_methods_layers.json"
2418: data = json.load(open(config_file))
2419:
2420: print(f'Total methods: {len(data["methods"])}')
2421: print('All have 8 layers:', all(len(m["layers"]) == 8 for m in data["methods"].values()))
2422: weight_check = all(abs(sum(m["weights"].values()) + sum(m["interaction_weights"].values()) - 1.0) < 0.01 for m in data["methods"].values())
2423: print('All weights sum to 1.0:', weight_check)
2424:
2425: for mid, m in list(data["methods"].items())[:3]:
2426:     print(f'\nSample: {mid.split(".")[-1]}')
2427:     print(f'  Layers: {len(m["layers"])}')
2428:     print(f'  Weights sum: {sum(m["weights"].values()) + sum(m["interaction_weights"].values()):.3f}')
2429:
2430:
2431:
2432: =====
2433: FILE: src/farfan_pipeline/__init__.py
2434: =====
2435:
2436: """Farfan Core Package Initialization.
2437:
2438: DEPRECATED: get_parameter_loader() is deprecated.
2439: Use ParameterLoaderV2 directly: from farfan_pipeline.core.parameters import ParameterLoaderV2
2440: """
2441:
2442: from farfan_pipeline.core.calibration.parameter_loader import ParameterLoader
2443: from farfan_pipeline.core.parameters import ParameterLoaderV2
2444:
2445: _parameter_loader = None
2446:
2447:
2448: def get_parameter_loader() -> "ParameterLoader":
2449:     """
2450:     DEPRECATED: Use ParameterLoaderV2.get(method_id, param_name) instead.
2451:
2452:     This function is kept for backward compatibility only.
2453:     Singleton global - guarantees EVERYONE uses the same one.
2454:     """
2455:     global _parameter_loader
2456:
2457:     if _parameter_loader is None:
2458:         _parameter_loader = ParameterLoader()
2459:         _parameter_loader.load()
2460:
2461:     return _parameter_loader
2462:
2463:
2464: __all__ = ["get_parameter_loader", "ParameterLoaderV2"]
```

```
2465:
2466:
2467:
2468: =====
2469: FILE: src/farfan_pipeline/analysis/Analyzer_one.py
2470: =====
2471:
2472: """
2473: Enhanced Municipal Development Plan Analyzer - Production-Grade Implementation.
2474:
2475: This module implements state-of-the-art techniques for comprehensive municipal plan analysis:
2476: - Semantic cubes with knowledge graphs and ontological reasoning
2477: - Multi-dimensional baseline analysis with automated extraction
2478: - Advanced NLP for multimodal text mining and causal discovery
2479: - Real-time monitoring with statistical process control
2480: - Bayesian optimization for resource allocation
2481: - Uncertainty quantification with Monte Carlo methods
2482:
2483: Python 3.11+ Compatible Version
2484: """
2485:
2486: from __future__ import annotations
2487:
2488: import hashlib
2489: import json
2490: import logging
2491: import re
2492: import time
2493: import warnings
2494: from collections import Counter, defaultdict
2495: from dataclasses import dataclass
2496: from datetime import datetime
2497: from pathlib import Path
2498: from typing import (
2499:     TYPE_CHECKING,
2500:     Any,
2501: )
2502:
2503: # from farfan_pipeline import get_parameter_loader # CALIBRATION DISABLED
2504: from farfan_pipeline.core.calibration.decorators import calibrated_method
2505:
2506: if TYPE_CHECKING:
2507:     from farfan_pipeline.utils.method_config_loader import MethodConfigLoader
2508:
2509: warnings.filterwarnings('ignore')
2510:
2511: # Constants
2512: SAMPLE_MUNICIPAL_PLAN = "sample_municipal_plan.txt"
2513: RANDOM_SEED = 42
2514:
2515: # Logging setup
2516: logging.basicConfig(level=logging.INFO)
2517: logger = logging.getLogger(__name__)
2518:
2519: # Missing imports for sklearn, nltk, numpy, pandas
2520: try:
```

```

2521:     import numpy as np
2522:     import pandas as pd
2523:     from nltk.corpus import stopwords
2524:     from nltk.tokenize import sent_tokenize
2525:     from sklearn.ensemble import IsolationForest
2526:     from sklearn.feature_extraction.text import TfidfVectorizer
2527: except ImportError as e:
2528:     logger.warning(f"Missing dependency: {e}")
2529:     # Provide fallbacks
2530:     TfidfVectorizer = None
2531:     IsolationForest = None
2532:     np = None
2533:     pd = None
2534:     sent_tokenize = None
2535:     stopwords = None
2536:
2537: # -----
2538: # 1. CORE DATA STRUCTURES
2539: # -----
2540:
2541: @dataclass
2542: class ValueChainLink:
2543:     """Represents a link in the municipal development value chain."""
2544:     name: str
2545:     instruments: list[str]
2546:     mediators: list[str]
2547:     outputs: list[str]
2548:     outcomes: list[str]
2549:     bottlenecks: list[str]
2550:     lead_time_days: float
2551:     conversion_rates: dict[str, float]
2552:     capacity_constraints: dict[str, float]
2553:
2554: class MunicipalOntology:
2555:     """Core ontology for municipal development domains."""
2556:
2557:     def __init__(self) -> None:
2558:         self.value_chain_links = {
2559:             "diagnostic_planning": ValueChainLink(
2560:                 name="diagnostic_planning",
2561:                 instruments=["territorial_diagnosis", "stakeholder_mapping", "needs_assessment"],
2562:                 mediators=["technical_capacity", "participatory_processes", "information_systems"],
2563:                 outputs=["diagnostic_report", "territorial_profile", "stakeholder_matrix"],
2564:                 outcomes=["shared_territorial_vision", "prioritized_problems"],
2565:                 bottlenecks=["data_availability", "technical_capacity_gaps", "time_constraints"],
2566:                 lead_time_days=90,
2567:                 conversion_rates={"diagnosis_to_strategy": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L95_59", 0.75)},
2568:                 capacity_constraints={"technical_staff": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L96_57", 0.8), "financial_resources": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L96_85", 0.6)},
2569:             ),
2570:             "strategic_planning": ValueChainLink(
2571:                 name="strategic_planning",
2572:                 instruments=["strategic_framework", "theory_of_change", "results_matrix"],
2573:                 mediators=["planning_methodology", "stakeholder_participation", "technical_assistance"],
2574:                 outputs=["development_plan", "sector_strategies", "investment_plan"],

```

```
2575:         outcomes=["strategic_alignment", "resource_optimization", "implementation_readiness"],
2576:         bottlenecks=["political_changes", "resource_constraints", "coordination_failures"],
2577:         lead_time_days=120,
2578:         conversion_rates={"strategy_to_programs": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_
L106_58", 0.80)},
2579:         capacity_constraints={"planning_expertise": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_para
m_L107_60", 0.7), "resources": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L107_78", 0.8)}
2580:     ),
2581:     "implementation": ValueChainLink(
2582:         name="implementation",
2583:         instruments=["project_management", "service_delivery", "capacity_building"],
2584:         mediators=["administrative_systems", "human_resources", "quality_control"],
2585:         outputs=["services_delivered", "capacities_developed", "results_achieved"],
2586:         outcomes=["improved_living_conditions", "enhanced_capabilities", "social_cohesion"],
2587:         bottlenecks=["budget_execution", "capacity_constraints", "coordination_failures"],
2588:         lead_time_days=365,
2589:         conversion_rates={"inputs_to_outputs": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L11
7_55", 0.75)},
2590:         capacity_constraints={"implementation_capacity": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto
_param_L118_65", 0.65), "coordination": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalOntology.__init__", "auto_param_L118_87", 0.60)}
2591:     )
2592: }
2593:
2594: self.policy_domains = {
2595:     "economic_development": ["competitiveness", "entrepreneurship", "employment"],
2596:     "social_development": ["education", "health", "housing"],
2597:     "territorial_development": ["land_use", "infrastructure", "connectivity"],
2598:     "institutional_development": ["governance", "transparency", "capacity_building"]
2599: }
2600:
2601: self.cross_cutting_themes = {
2602:     "governance": ["transparency", "accountability", "participation"],
2603:     "equity": ["gender_equality", "social_inclusion", "poverty_reduction"],
2604:     "sustainability": ["environmental_protection", "climate_adaptation"],
2605:     "innovation": ["digital_transformation", "process_innovation"]
2606: }
2607:
2608: # -----
2609: # 2. SEMANTIC ANALYSIS ENGINE
2610: # -----
2611:
2612: class SemanticAnalyzer:
2613:     """Advanced semantic analysis for municipal documents."""
2614:
2615:     def __init__(
2616:         self,
2617:         ontology: MunicipalOntology,
2618:         config_loader: MethodConfigLoader | None = None,
2619:         max_features: int | None = None,
2620:         ngram_range: tuple[int, int] | None = None,
2621:         similarity_threshold: float | None = None
2622:     ) -> None:
2623:         """
2624:         Initialize SemanticAnalyzer.
2625:
2626:         Args:
```

```
2627:         ontology: Municipal ontology for semantic classification
2628:         config_loader: Optional MethodConfigLoader for canonical parameter access
2629:         max_features: TF-IDF max features (overrides config_loader)
2630:         ngram_range: N-gram range for feature extraction (overrides config_loader)
2631:         similarity_threshold: Similarity threshold for concept detection (overrides config_loader)
2632:     """
2633:     self.ontology = ontology
2634:
2635:     # Load parameters from canonical JSON if config_loader provided
2636:     if config_loader is not None:
2637:         try:
2638:             if max_features is None:
2639:                 max_features = config_loader.get_method_parameter(
2640:                     "ANLZ.SA.extract_cube_v1", "max_features"
2641:                 )
2642:             if ngram_range is None:
2643:                 ngram_range = tuple(config_loader.get_method_parameter(
2644:                     "ANLZ.SA.extract_cube_v1", "ngram_range"
2645:                 ))
2646:             if similarity_threshold is None:
2647:                 similarity_threshold = config_loader.get_method_parameter(
2648:                     "ANLZ.SA.extract_cube_v1", "similarity_threshold"
2649:                 )
2650:         except (KeyError, AttributeError) as e:
2651:             logger.warning(f"Failed to load parameters from config_loader: {e}. Using defaults.")
2652:
2653:     # Use defaults if not provided
2654:     self.max_features = max_features if max_features is not None else 1000
2655:     self.ngram_range = ngram_range if ngram_range is not None else (1, 3)
2656:     self.similarity_threshold = similarity_threshold if similarity_threshold is not None else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.M
unicipalOntology.__init__", "auto_param_L184_98", 0.3)
2657:
2658:     if TfidfVectorizer is not None:
2659:         self.vectorizer = TfidfVectorizer(
2660:             max_features=self.max_features,
2661:             stop_words='english',
2662:             ngram_range=self.ngram_range
2663:         )
2664:     else:
2665:         self.vectorizer = None
2666:
2667: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer.extract_semantic_cube")
2668: def extract_semantic_cube(self, document_segments: list[str]) -> dict[str, Any]:
2669:     """Extract multidimensional semantic cube from document segments."""
2670:
2671:     if not document_segments:
2672:         return self._empty_semantic_cube()
2673:
2674:     # Vectorize segments
2675:     segment_vectors = self._vectorize_segments(document_segments)
2676:
2677:     # Initialize semantic cube
2678:     semantic_cube = {
2679:         "dimensions": {
2680:             "value_chain_links": defaultdict(list),
2681:             "policy_domains": defaultdict(list),
```

```
2682:         "cross_cutting_themes": defaultdict(list)
2683:     },
2684:     "measures": {
2685:         "semantic_density": [],
2686:         "coherence_scores": [],
2687:         "complexity_metrics": []
2688:     },
2689:     "metadata": {
2690:         "extraction_timestamp": datetime.now().isoformat(),
2691:         "total_segments": len(document_segments),
2692:         "processing_parameters": {}
2693:     }
2694: }
2695:
2696: # Process each segment
2697: for idx, segment in enumerate(document_segments):
2698:     segment_data = self._process_segment(segment, idx, segment_vectors[idx])
2699:
2700:     # Classify by value chain links
2701:     link_scores = self._classify_value_chain_link(segment)
2702:     for link, score in link_scores.items():
2703:         if score > self.similarity_threshold: # Configurable threshold for inclusion
2704:             semantic_cube["dimensions"]["value_chain_links"][link].append(segment_data)
2705:
2706:     # Classify by policy domains
2707:     domain_scores = self._classify_policy_domain(segment)
2708:     for domain, score in domain_scores.items():
2709:         if score > self.similarity_threshold:
2710:             semantic_cube["dimensions"]["policy_domains"][domain].append(segment_data)
2711:
2712:     # Extract cross-cutting themes
2713:     theme_scores = self._classify_cross_cutting_themes(segment)
2714:     for theme, score in theme_scores.items():
2715:         if score > self.similarity_threshold:
2716:             semantic_cube["dimensions"]["cross_cutting_themes"][theme].append(segment_data)
2717:
2718:     # Add measures
2719:     semantic_cube["measures"]["semantic_density"].append(segment_data["semantic_density"])
2720:     semantic_cube["measures"]["coherence_scores"].append(segment_data["coherence_score"])
2721:
2722: # Calculate aggregate measures
2723: if semantic_cube["measures"]["coherence_scores"]:
2724:     if np is not None:
2725:         semantic_cube["measures"]["overall_coherence"] = np.mean(
2726:             semantic_cube["measures"]["coherence_scores"]
2727:         )
2728:     else:
2729:         semantic_cube["measures"]["overall_coherence"] = sum(
2730:             semantic_cube["measures"]["coherence_scores"]
2731:         ) / len(semantic_cube["measures"]["coherence_scores"])
2732: else:
2733:     semantic_cube["measures"]["overall_coherence"] = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer.extract_semantic_cube", "auto_param_L261_61", 0.0)
2734:
2735: semantic_cube["measures"]["semantic_complexity"] = self._calculate_semantic_complexity(semantic_cube)
2736:
```

```
2737:         logger.info(f"Extracted semantic cube from {len(document_segments)} segments")
2738:         return semantic_cube
2739:
2740: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._empty_semantic_cube")
2741: def _empty_semantic_cube(self) -> dict[str, Any]:
2742:     """Return empty semantic cube structure."""
2743:     return {
2744:         "dimensions": {
2745:             "value_chain_links": {},
2746:             "policy_domains": {},
2747:             "cross_cutting_themes": {}
2748:         },
2749:         "measures": {
2750:             "semantic_density": [],
2751:             "coherence_scores": [],
2752:             "overall_coherence": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._empty_semantic_cube", "auto_param_L280_37",
0.0),
2753:             "semantic_complexity": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._empty_semantic_cube", "auto_param_L281_39"
, 0.0)
2754:         },
2755:         "metadata": {
2756:             "extraction_timestamp": datetime.now().isoformat(),
2757:             "total_segments": 0,
2758:             "processing_parameters": {}
2759:         }
2760:     }
2761:
2762: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._vectorize_segments")
2763: def _vectorize_segments(self, segments: list[str]) -> np.ndarray:
2764:     """Vectorize document segments using TF-IDF."""
2765:     if self.vectorizer is not None:
2766:         try:
2767:             return self.vectorizer.fit_transform(segments).toarray()
2768:         except Exception as e:
2769:             logger.warning(f"Vectorization failed: {e}")
2770:
2771:     # Fallback
2772:     if np is not None:
2773:         return np.zeros((len(segments), 100))
2774:     else:
2775:         # Return list of lists if numpy is not available
2776:         return [[ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._vectorize_segments", "auto_param_L304_21", 0.0)] * 100 for _
in range(len(segments))]
2777:
2778: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._process_segment")
2779: def _process_segment(self, segment: str, idx: int, vector) -> dict[str, Any]:
2780:     """Process individual segment and extract features."""
2781:
2782:     # Basic text statistics
2783:     words = segment.split()
2784:
2785:     # Calculate sentence count
2786:     if sent_tokenize is not None:
2787:         try:
2788:             sentences = sent_tokenize(segment)
2789:         except:
```

```
2790:         # Fallback to simple splitting
2791:         sentences = [s.strip() for s in re.split(r'[!?!]+', segment) if len(s.strip()) > 10]
2792:     else:
2793:         # Fallback to simple splitting
2794:         sentences = [s.strip() for s in re.split(r'[!?!]+', segment) if len(s.strip()) > 10]
2795:
2796:         # Calculate semantic density (simplified)
2797:         semantic_density = len(set(words)) / len(words) if words else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._process_seg
ment", "auto_param_L325_70", 0.0)
2798:
2799:         # Calculate coherence score (simplified)
2800:         coherence_score = min(ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._process_segment", "auto_param_L328_30", 1.0), len(s
entences) / 10) if sentences else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._process_segment", "auto_param_L328_74", 0.0)
2801:
2802:         # Convert vector to list if it's a numpy array
2803:         if np is not None and isinstance(vector, np.ndarray):
2804:             vector = vector.tolist()
2805:
2806:         return {
2807:             "segment_id": idx,
2808:             "text": segment,
2809:             "vector": vector,
2810:             "word_count": len(words),
2811:             "sentence_count": len(sentences),
2812:             "semantic_density": semantic_density,
2813:             "coherence_score": coherence_score
2814:         }
2815:
2816: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_value_chain_link")
2817: def _classify_value_chain_link(self, segment: str) -> dict[str, float]:
2818:     """Classify segment by value chain link using keyword matching."""
2819:     link_scores = {}
2820:     segment_lower = segment.lower()
2821:
2822:     for link_name, link_obj in self.ontology.value_chain_links.items():
2823:         score = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_value_chain_link", "score", 0.0) # Refactored
2824:         total_keywords = 0
2825:
2826:         # Check all link components
2827:         all_keywords = (link_obj.instruments + link_obj.mediators +
2828:             link_obj.outputs + link_obj.outcomes)
2829:
2830:         for keyword in all_keywords:
2831:             total_keywords += 1
2832:             if keyword.lower().replace("_", " ") in segment_lower:
2833:                 score += ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_value_chain_link", "auto_param_L361_29", 1.
0)
2834:
2835:         # Normalize score
2836:         link_scores[link_name] = score / total_keywords if total_keywords > 0 else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnal
yzer._classify_value_chain_link", "auto_param_L364_87", 0.0)
2837:
2838:     return link_scores
2839:
2840: @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_policy_domain")
2841: def _classify_policy_domain(self, segment: str) -> dict[str, float]:
```

```
2842:         """Classify segment by policy domain using keyword matching."""
2843:         domain_scores = {}
2844:         segment_lower = segment.lower()
2845:
2846:         for domain, keywords in self.ontology.policy_domains.items():
2847:             score = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_policy_domain", "score", 0.0) # Refactored
2848:             for keyword in keywords:
2849:                 if keyword.lower() in segment_lower:
2850:                     score += ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_policy_domain", "auto_param_L378_29", 1.0)
2851:
2852:             domain_scores[domain] = score / len(keywords) if keywords else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classi
fy_policy_domain", "auto_param_L380_75", 0.0)
2853:
2854:         return domain_scores
2855:
2856:     @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_cross_cutting_themes")
2857:     def _classify_cross_cutting_themes(self, segment: str) -> dict[str, float]:
2858:         """Classify segment by cross-cutting themes."""
2859:         theme_scores = {}
2860:         segment_lower = segment.lower()
2861:
2862:         for theme, keywords in self.ontology.cross_cutting_themes.items():
2863:             score = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_cross_cutting_themes", "score", 0.0) # Refactored
2864:             for keyword in keywords:
2865:                 if keyword.lower().replace("_", " ") in segment_lower:
2866:                     score += ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify_cross_cutting_themes", "auto_param_L394_29"
, 1.0)
2867:
2868:             theme_scores[theme] = score / len(keywords) if keywords else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._classify
_cross_cutting_themes", "auto_param_L396_73", 0.0)
2869:
2870:         return theme_scores
2871:
2872:     @calibrated_method("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._calculate_semantic_complexity")
2873:     def _calculate_semantic_complexity(self, semantic_cube: dict[str, Any]) -> float:
2874:         """Calculate semantic complexity of the cube."""
2875:
2876:         # Count unique concepts across dimensions
2877:         unique_concepts = set()
2878:         for dimension_data in semantic_cube["dimensions"].values():
2879:             for category in dimension_data:
2880:                 unique_concepts.add(category)
2881:
2882:         # Normalize complexity
2883:         max_expected_concepts = 20
2884:         return min(ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.SemanticAnalyzer._calculate_semantic_complexity", "auto_param_L412_19", 1.0), le
n(unique_concepts) / max_expected_concepts)
2885:
2886: # -----
2887: # 3. PERFORMANCE ANALYZER
2888: # -----
2889:
2890: class PerformanceAnalyzer:
2891:     """Analyze value chain performance with operational loss functions."""
2892:
2893:     def __init__(self, ontology: MunicipalOntology) -> None:
```

```
2894:         self.ontology = ontology
2895:         if IsolationForest is not None:
2896:             self.bottleneck_detector = IsolationForest(contamination=ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer.__init__",
"auto_param_L424_69", 0.1), random_state=RANDOM_SEED)
2897:         else:
2898:             self.bottleneck_detector = None
2899:
2900:     @calibrated_method("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer.analyze_performance")
2901:     def analyze_performance(self, semantic_cube: dict[str, Any]) -> dict[str, Any]:
2902:         """Analyze performance indicators across value chain links."""
2903:
2904:         performance_analysis = {
2905:             "value_chain_metrics": {},
2906:             "bottleneck_analysis": {},
2907:             "operational_loss_functions": {},
2908:             "optimization_recommendations": []
2909:         }
2910:
2911:         # Analyze each value chain link
2912:         for link_name, link_config in self.ontology.value_chain_links.items():
2913:             link_segments = semantic_cube["dimensions"]["value_chain_links"].get(link_name, [])
2914:
2915:             # Calculate metrics
2916:             metrics = self._calculate_throughput_metrics(link_segments, link_config)
2917:             bottlenecks = self._detect_bottlenecks(link_segments, link_config)
2918:             loss_functions = self._calculate_loss_functions(metrics, link_config)
2919:
2920:             performance_analysis["value_chain_metrics"][link_name] = metrics
2921:             performance_analysis["bottleneck_analysis"][link_name] = bottlenecks
2922:             performance_analysis["operational_loss_functions"][link_name] = loss_functions
2923:
2924:         # Generate recommendations
2925:         performance_analysis["optimization_recommendations"] = self._generate_recommendations(
2926:             performance_analysis
2927:         )
2928:
2929:         logger.info(f"Performance analysis completed for {len(performance_analysis['value_chain_metrics'])} links")
2930:         return performance_analysis
2931:
2932:     @calibrated_method("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_throughput_metrics")
2933:     def _calculate_throughput_metrics(self, segments: list[dict], link_config: ValueChainLink) -> dict[str, Any]:
2934:         """Calculate throughput metrics for a value chain link."""
2935:
2936:         if not segments:
2937:             return {
2938:                 "throughput": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_throughput_metrics", "auto_param_L466_
30", 0.0),
2939:                 "efficiency_score": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_throughput_metrics", "auto_param
_L467_36", 0.0),
2940:                 "capacity_utilization": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_throughput_metrics", "auto_p
aram_L468_40", 0.0)
2941:             }
2942:
2943:         # Calculate semantic throughput
2944:         total_semantic_content = sum(seg["semantic_density"] for seg in segments)
2945:
```

```
2946:         if np is not None:
2947:             avg_coherence = np.mean([seg["coherence_score"] for seg in segments])
2948:         else:
2949:             avg_coherence = sum(seg["coherence_score"] for seg in segments) / len(segments)
2950:
2951:         # Capacity utilization
2952:         theoretical_max_segments = 50
2953:         capacity_utilization = len(segments) / theoretical_max_segments
2954:
2955:         # Efficiency score
2956:         efficiency_score = (total_semantic_content / len(segments)) * avg_coherence
2957:
2958:         # Throughput calculation
2959:         if np is not None:
2960:             throughput = len(segments) * avg_coherence * np.mean(list(link_config.conversion_rates.values()))
2961:         else:
2962:             throughput = len(segments) * avg_coherence * sum(link_config.conversion_rates.values()) / len(link_config.conversion_rates)
2963:
2964:         return {
2965:             "throughput": float(throughput),
2966:             "efficiency_score": float(efficiency_score),
2967:             "capacity_utilization": float(capacity_utilization),
2968:             "segment_count": len(segments)
2969:         }
2970:
2971: @calibrated_method("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottlenecks")
2972: def _detect_bottlenecks(self, segments: list[dict], link_config: ValueChainLink) -> dict[str, Any]:
2973:     """Detect bottlenecks in value chain link."""
2974:
2975:     bottleneck_analysis = {
2976:         "capacity_constraints": {},
2977:         "bottleneck_scores": {}
2978:     }
2979:
2980:     # Analyze capacity constraints
2981:     for constraint_type, constraint_value in link_config.capacity_constraints.items():
2982:         if constraint_value < ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottlenecks", "auto_param_L510_34", 0
.7):
2983:             bottleneck_analysis["capacity_constraints"][constraint_type] = {
2984:                 "current_capacity": constraint_value,
2985:                 "severity": "high" if constraint_value < ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottleneck
s", "auto_param_L513_61", 0.5) else "medium"
2986:             }
2987:
2988:     # Calculate bottleneck scores
2989:     for bottleneck_type in link_config.bottlenecks:
2990:         score = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottlenecks", "score", 0.0) # Refactored
2991:         if segments:
2992:             # Count mentions of bottleneck in segments
2993:             mentions = sum(
2994:                 1 for seg in segments
2995:                 if bottleneck_type.replace("_", " ").lower() in seg["text"].lower()
2996:             )
2997:             score = mentions / len(segments)
2998:
2999:         bottleneck_analysis["bottleneck_scores"][bottleneck_type] = {
```

```
3000:         "score": score,
3001:         "severity": "high" if score > ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottlenecks", "auto_param_L529_46", 0.2) else "medium" if score > ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._detect_bottlenecks", "auto_param_L529_75", 0.1) else "low"
3002:     }
3003:
3004:     return bottleneck_analysis
3005:
3006: @calibrated_method("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions")
3007: def _calculate_loss_functions(self, metrics: dict[str, Any], link_config: ValueChainLink) -> dict[str, Any]:
3008:     """Calculate operational loss functions."""
3009:
3010:     # Throughput loss (quadratic)
3011:     target_throughput = 5.0
3012:     throughput_gap = max(0, target_throughput - metrics["throughput"])
3013:     throughput_loss = throughput_gap ** 2
3014:
3015:     # Efficiency loss (exponential)
3016:     target_efficiency = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions", "target_efficiency", 0.1)
3017:     efficiency_gap = max(0, target_efficiency - metrics["efficiency_score"])
3018:
3019:     if np is not None:
3020:         efficiency_loss = np.exp(efficiency_gap * 2) - 1
3021:     else:
3022:         # Approximate exponential function
3023:         efficiency_loss = (1 + efficiency_gap) ** 2 - 1
3024:
3025:     # Time loss (linear)
3026:     baseline_time = link_config.lead_time_days
3027:     capacity_utilization = metrics["capacity_utilization"]
3028:     time_multiplier = 1 + (1 - capacity_utilization) * ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions", "auto_param_L556_59", 0.5)
3029:     time_loss = baseline_time * (time_multiplier - 1)
3030:
3031:     # Composite loss
3032:     composite_loss = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions", "auto_param_L560_25", 0.4)
3033:     * throughput_loss + ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions", "auto_param_L560_49", 0.4) * efficiency_loss + ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._calculate_loss_functions", "auto_param_L560_73", 0.2) * time_loss
3034:
3035:     return {
3036:         "throughput_loss": float(throughput_loss),
3037:         "efficiency_loss": float(efficiency_loss),
3038:         "time_loss": float(time_loss),
3039:         "composite_loss": float(composite_loss)
3040:     }
3041:
3042: @calibrated_method("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._generate_recommendations")
3043: def _generate_recommendations(self, performance_analysis: dict[str, Any]) -> list[dict[str, Any]]:
3044:     """Generate optimization recommendations."""
3045:
3046:     recommendations = []
3047:
3048:     for link_name, metrics in performance_analysis["value_chain_metrics"].items():
3049:         if metrics["efficiency_score"] < ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.PerformanceAnalyzer._generate_recommendations", "auto_param_L576_45", 0.5):
```

```
3049:         recommendations.append({
3050:             "link": link_name,
3051:             "type": "efficiency_improvement",
3052:             "priority": "high",
3053:             "description": f"Critical efficiency improvement needed for {link_name}"
3054:         })
3055:
3056:     if metrics["throughput"] < 20:
3057:         recommendations.append({
3058:             "link": link_name,
3059:             "type": "throughput_optimization",
3060:             "priority": "medium",
3061:             "description": f"Throughput optimization required for {link_name}"
3062:         })
3063:
3064:     return recommendations
3065:
3066: # -----
3067: # 4. TEXT MINING ENGINE
3068: # -----
3069:
3070: class TextMiningEngine:
3071:     """Advanced text mining for critical diagnosis."""
3072:
3073:     def __init__(self, ontology: MunicipalOntology) -> None:
3074:         self.ontology = ontology
3075:
3076:         # Initialize simple keyword extractor
3077:         self.stop_words = set()
3078:         if stopwords is not None:
3079:             try:
3080:                 self.stop_words = set(stopwords.words('spanish'))
3081:             except LookupError:
3082:                 # Download if not available
3083:                 try:
3084:                     import nltk
3085:                     nltk.download('stopwords')
3086:                     self.stop_words = set(stopwords.words('spanish'))
3087:                 except:
3088:                     logger.warning("Could not download NLTK stopwords. Using empty set.")
3089:
3090: @calibrated_method("farfan_core.analysis.Analyzer_one.TextMiningEngine.diagnose_critical_links")
3091: def diagnose_critical_links(self, semantic_cube: dict[str, Any],
3092:                             performance_analysis: dict[str, Any]) -> dict[str, Any]:
3093:     """Diagnose critical value chain links."""
3094:
3095:     diagnosis_results = {
3096:         "critical_links": {},
3097:         "risk_assessment": {},
3098:         "intervention_recommendations": {}
3099:     }
3100:
3101:     # Identify critical links
3102:     critical_links = self._identify_critical_links(performance_analysis)
3103:
3104:     # Analyze each critical link
```

```
3105:         for link_name, criticality_score in critical_links.items():
3106:             link_segments = semantic_cube["dimensions"]["value_chain_links"].get(link_name, [])
3107:
3108:             # Text analysis
3109:             text_analysis = self._analyze_link_text(link_segments)
3110:
3111:             # Risk assessment
3112:             risk_assessment = self._assess_risks(link_segments, text_analysis)
3113:
3114:             # Intervention recommendations
3115:             interventions = self._generate_interventions(link_name, risk_assessment, text_analysis)
3116:
3117:             diagnosis_results["critical_links"][link_name] = {
3118:                 "criticality_score": criticality_score,
3119:                 "text_analysis": text_analysis
3120:             }
3121:             diagnosis_results["risk_assessment"][link_name] = risk_assessment
3122:             diagnosis_results["intervention_recommendations"][link_name] = interventions
3123:
3124:             logger.info(f"Diagnosed {len(critical_links)} critical links")
3125:             return diagnosis_results
3126:
3127: @calibrated_method("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links")
3128: def _identify_critical_links(self, performance_analysis: dict[str, Any]) -> dict[str, float]:
3129:     """Identify critical links based on performance metrics."""
3130:
3131:     critical_links = {}
3132:
3133:     for link_name, metrics in performance_analysis["value_chain_metrics"].items():
3134:         criticality_score = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "criticality_score", 0.
0) # Refactored
3135:
3136:         # Low efficiency indicates criticality
3137:         if metrics["efficiency_score"] < ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L665_45", 0.5):
3138:             criticality_score += ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L666_37", 0.4)
3139:
3140:         # Low throughput indicates criticality
3141:         if metrics["throughput"] < 20:
3142:             criticality_score += ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L670_37", 0.3)
3143:
3144:         # High loss functions indicate criticality
3145:         if link_name in performance_analysis["operational_loss_functions"]:
3146:             loss = performance_analysis["operational_loss_functions"][link_name]["composite_loss"]
3147:             normalized_loss = min(ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L675_38", 1.0), loss / 100)
3148:             criticality_score += normalized_loss * ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L676_55", 0.3)
3149:
3150:         if criticality_score > ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.TextMiningEngine._identify_critical_links", "auto_param_L678_35", 0.4):
3151:             critical_links[link_name] = criticality_score
3152:
3153:     return critical_links
```

```
3154:
3155:     @calibrated_method("farfan_core.analysis.Analyzer_one.TextMiningEngine._analyze_link_text")
3156:     def _analyze_link_text(self, segments: list[dict]) -> dict[str, Any]:
3157:         """Analyze text content for a link."""
3158:
3159:         if not segments:
3160:             return {"word_count": 0, "keywords": [], "sentiment": "neutral"}
3161:
3162:         # Combine all text
3163:         combined_text = " ".join([seg["text"] for seg in segments])
3164:         words = [word.lower() for word in combined_text.split()]
3165:         if word.lower() not in self.stop_words and len(word) > 2]
3166:
3167:         # Extract keywords
3168:         word_freq = Counter(words)
3169:         keywords = [word for word, count in word_freq.most_common(10)]
3170:
3171:         # Simple sentiment analysis
3172:         positive_words = ['bueno', 'excelente', 'positivo', 'lograr', 'Ã©xito']
3173:         negative_words = ['problema', 'dificultad', 'limitaciÃ³n', 'falta', 'dÃ©ficit']
3174:
3175:         positive_count = sum(1 for word in words if word in positive_words)
3176:         negative_count = sum(1 for word in words if word in negative_words)
3177:
3178:         if positive_count > negative_count:
3179:             sentiment = "positive"
3180:         elif negative_count > positive_count:
3181:             sentiment = "negative"
3182:         else:
3183:             sentiment = "neutral"
3184:
3185:         return {
3186:             "word_count": len(words),
3187:             "keywords": keywords,
3188:             "sentiment": sentiment,
3189:             "positive_indicators": positive_count,
3190:             "negative_indicators": negative_count
3191:         }
3192:
3193:     @calibrated_method("farfan_core.analysis.Analyzer_one.TextMiningEngine._assess_risks")
3194:     def _assess_risks(self, segments: list[dict], text_analysis: dict[str, Any]) -> dict[str, Any]:
3195:         """Assess risks for a value chain link."""
3196:
3197:         risk_assessment = {
3198:             "overall_risk": "low",
3199:             "risk_factors": []
3200:         }
3201:
3202:         # Sentiment-based risk
3203:         if text_analysis["sentiment"] == "negative":
3204:             risk_assessment["risk_factors"].append("Negative sentiment detected")
3205:
3206:         # Content-based risk
3207:         if text_analysis["negative_indicators"] > 3:
3208:             risk_assessment["risk_factors"].append("High frequency of negative indicators")
3209:
```

```
3210:         # Volume-based risk
3211:         if text_analysis["word_count"] < 50:
3212:             risk_assessment["risk_factors"].append("Limited content volume")
3213:
3214:         # Overall risk level
3215:         if len(risk_assessment["risk_factors"]) > 2:
3216:             risk_assessment["overall_risk"] = "high"
3217:         elif len(risk_assessment["risk_factors"]) > 0:
3218:             risk_assessment["overall_risk"] = "medium"
3219:
3220:         return risk_assessment
3221:
3222: @calibrated_method("farfan_core.analysis.Analyzer_one.TextMiningEngine._generate_interventions")
3223: def _generate_interventions(self, link_name: str, risk_assessment: dict[str, Any],
3224:                             text_analysis: dict[str, Any]) -> list[dict[str, str]]:
3225:     """Generate intervention recommendations."""
3226:
3227:     interventions = []
3228:
3229:     if risk_assessment["overall_risk"] == "high":
3230:         interventions.append({
3231:             "type": "immediate",
3232:             "description": f"Priority intervention required for {link_name}",
3233:             "timeline": "1-3 months"
3234:         })
3235:
3236:     if text_analysis["sentiment"] == "negative":
3237:         interventions.append({
3238:             "type": "stakeholder_engagement",
3239:             "description": "Address concerns through stakeholder engagement",
3240:             "timeline": "ongoing"
3241:         })
3242:
3243:     if text_analysis["word_count"] < 50:
3244:         interventions.append({
3245:             "type": "documentation",
3246:             "description": "Improve documentation and content development",
3247:             "timeline": "3-6 months"
3248:         })
3249:
3250:     return interventions
3251:
3252: # -----
3253: # 5. COMPREHENSIVE ANALYZER
3254: # -----
3255:
3256: class MunicipalAnalyzer:
3257:     """Main analyzer integrating all components."""
3258:
3259:     def __init__(self) -> None:
3260:         self.ontology = MunicipalOntology()
3261:         self.semantic_analyzer = SemanticAnalyzer(self.ontology)
3262:         self.performance_analyzer = PerformanceAnalyzer(self.ontology)
3263:         self.text_miner = TextMiningEngine(self.ontology)
3264:
3265:         logger.info("MunicipalAnalyzer initialized successfully")
```

```
3266:
3267:     @calibrated_method("farfan_core.analysis.Analyzer_one.MunicipalAnalyzer.analyze_document")
3268:     def analyze_document(self, document_path: str) -> dict[str, Any]:
3269:         """Perform comprehensive analysis of a municipal document."""
3270:
3271:         start_time = time.time()
3272:         logger.info(f"Starting analysis of {document_path}")
3273:
3274:         try:
3275:             # Load and process document
3276:             document_segments = self._load_document(document_path)
3277:
3278:             # Semantic analysis
3279:             logger.info("Performing semantic analysis...")
3280:             semantic_cube = self.semantic_analyzer.extract_semantic_cube(document_segments)
3281:
3282:             # Performance analysis
3283:             logger.info("Analyzing performance indicators...")
3284:             performance_analysis = self.performance_analyzer.analyze_performance(semantic_cube)
3285:
3286:             # Text mining and diagnosis
3287:             logger.info("Performing text mining and diagnosis...")
3288:             critical_diagnosis = self.text_miner.diagnose_critical_links(
3289:                 semantic_cube, performance_analysis
3290:             )
3291:
3292:             # Compile results
3293:             results = {
3294:                 "document_path": document_path,
3295:                 "analysis_timestamp": datetime.now().isoformat(),
3296:                 "processing_time_seconds": time.time() - start_time,
3297:                 "semantic_cube": semantic_cube,
3298:                 "performance_analysis": performance_analysis,
3299:                 "critical_diagnosis": critical_diagnosis,
3300:                 "summary": self._generate_summary(semantic_cube, performance_analysis, critical_diagnosis)
3301:             }
3302:
3303:             logger.info(f"Analysis completed in {time.time() - start_time:.2f} seconds")
3304:             return results
3305:
3306:         except Exception as e:
3307:             logger.error(f"Analysis failed: {str(e)}")
3308:             raise
3309:
3310:     @calibrated_method("farfan_core.analysis.Analyzer_one.MunicipalAnalyzer._load_document")
3311:     def _load_document(self, document_path: str) -> list[str]:
3312:         """Load and segment document."""
3313:
3314:         # Delegate to factory for I/O operation
3315:         from farfan_pipeline.analysis.factory import read_text_file
3316:
3317:         content = read_text_file(document_path)
3318:
3319:         # Simple sentence segmentation
3320:         sentences = re.split(r'[.!?]+', content)
3321:
```

```
3322:         # Clean and filter segments
3323:         segments = []
3324:         for sentence in sentences:
3325:             cleaned = sentence.strip()
3326:             if len(cleaned) > 20 and not cleaned.startswith(('PÃgina', 'Page')):
3327:                 segments.append(cleaned)
3328:
3329:         return segments[:100] # Limit for processing efficiency
3330:
3331: @calibrated_method("farfan_core.analysis.Analyzer_one.MunicipalAnalyzer._generate_summary")
3332: def _generate_summary(self, semantic_cube: dict[str, Any],
3333:                       performance_analysis: dict[str, Any],
3334:                       critical_diagnosis: dict[str, Any]) -> dict[str, Any]:
3335:     """Generate executive summary of analysis."""
3336:
3337:     # Count dimensions
3338:     total_segments = semantic_cube["metadata"]["total_segments"]
3339:     value_chain_coverage = len(semantic_cube["dimensions"]["value_chain_links"])
3340:     policy_domain_coverage = len(semantic_cube["dimensions"]["policy_domains"])
3341:
3342:     # Performance summary
3343:     if performance_analysis["value_chain_metrics"]:
3344:         if np is not None:
3345:             avg_efficiency = np.mean([
3346:                 metrics["efficiency_score"]
3347:                 for metrics in performance_analysis["value_chain_metrics"].values()
3348:             ])
3349:         else:
3350:             avg_efficiency = sum(
3351:                 metrics["efficiency_score"]
3352:                 for metrics in performance_analysis["value_chain_metrics"].values()
3353:             ) / len(performance_analysis["value_chain_metrics"])
3354:     else:
3355:         avg_efficiency = ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.MunicipalAnalyzer._generate_summary", "avg_efficiency", 0.0) # Refacto
red
3356:
3357:     # Critical links count
3358:     critical_links_count = len(critical_diagnosis["critical_links"])
3359:
3360:     return {
3361:         "document_coverage": {
3362:             "total_segments_analyzed": total_segments,
3363:             "value_chain_links_identified": value_chain_coverage,
3364:             "policy_domains_covered": policy_domain_coverage
3365:         },
3366:         "performance_summary": {
3367:             "average_efficiency_score": float(avg_efficiency),
3368:             "recommendations_count": len(performance_analysis["optimization_recommendations"])
3369:         },
3370:         "risk_assessment": {
3371:             "critical_links_identified": critical_links_count,
3372:             "overall_risk_level": "high" if critical_links_count > 2 else "medium" if critical_links_count > 0 else "low"
3373:         }
3374:     }
3375:
3376: # -----
```

```
3377: # 6. EXAMPLE USAGE AND UTILITIES
3378: # -----
3379:
3380: def example_usage():
3381:     """Example usage of the Municipal Analyzer."""
3382:
3383:     # Initialize analyzer
3384:     analyzer = MunicipalAnalyzer()
3385:
3386:     # Create sample document
3387:     sample_text = """
3388:     El Plan de Desarrollo Municipal tiene como objetivo principal fortalecer
3389:     la capacidad institucional y mejorar la calidad de vida de los habitantes.
3390:
3391:     En el Ãrea de desarrollo econÃmico, se implementarÃn programas de
3392:     emprendimiento y competitividad empresarial. Los recursos asignados
3393:     permitirÃn crear 500 nuevos empleos en el sector productivo.
3394:
3395:     Para el desarrollo social, se priorizarÃn proyectos de educaciÃn y salud.
3396:     Se construirÃn 3 nuevos centros de salud y se mejorarÃn 10 instituciones
3397:     educativas. El presupuesto destinado asciende a 2.5 millones de pesos.
3398:
3399:     La estrategia de implementaciÃn incluye mecanismos de participaciÃn
3400:     ciudadana y seguimiento continuo a travÃs de indicadores de gestiÃn.
3401:     Se establecerÃn alianzas con el sector privado y organizaciones sociales.
3402:
3403:     Los principales riesgos identificados incluyen limitaciones presupuestales
3404:     y posibles cambios en el contexto polÃtico. Se requiere fortalecer
3405:     la coordinaciÃn interinstitucional para garantizar el Ãxito.
3406:     """
3407:
3408:     # Save sample to file
3409:     # Delegate to factory for I/O operation
3410:     from farfan_pipeline.analysis.factory import write_text_file
3411:
3412:     write_text_file(sample_text, SAMPLE_MUNICIPAL_PLAN)
3413:
3414:     try:
3415:         # Analyze document
3416:         results = analyzer.analyze_document(SAMPLE_MUNICIPAL_PLAN)
3417:
3418:         # Print summary
3419:         print("\n" + "=" * 60)
3420:         print("MUNICIPAL DEVELOPMENT PLAN ANALYSIS")
3421:         print("=" * 60)
3422:
3423:         print(f"\nDocument: {results['document_path']}")
3424:         print(f"Processing time: {results['processing_time_seconds']:.2f} seconds")
3425:
3426:         # Semantic analysis summary
3427:         print("\nSEMANTIC ANALYSIS:")
3428:         cube = results['semantic_cube']
3429:         print(f"- Total segments processed: {cube['metadata']['total_segments']}")
3430:         print(f"- Overall coherence: {cube['measures']['overall_coherence']:.2f}")
3431:         print(f"- Semantic complexity: {cube['measures']['semantic_complexity']:.2f}")
3432:
```

```
3433:     print("\nValue Chain Links Identified:")
3434:     for link, segments in cube['dimensions']['value_chain_links'].items():
3435:         print(f"    - {link}: {len(segments)} segments")
3436:
3437:     print("\nPolicy Domains Covered:")
3438:     for domain, segments in cube['dimensions']['policy_domains'].items():
3439:         print(f"    - {domain}: {len(segments)} segments")
3440:
3441:     # Performance analysis summary
3442:     print("\nPERFORMANCE ANALYSIS:")
3443:     perf = results['performance_analysis']
3444:     for link, metrics in perf['value_chain_metrics'].items():
3445:         print(f"\n{link.replace('_', ' ').title()}:")
3446:         print(f"    - Efficiency: {metrics['efficiency_score']:.2f}")
3447:         print(f"    - Throughput: {metrics['throughput']:.1f}")
3448:         print(f"    - Capacity utilization: {metrics['capacity_utilization']:.2f}")
3449:
3450:     print(f"\nOptimization Recommendations: {len(perf['optimization_recommendations'])}")
3451:     for rec in perf['optimization_recommendations'][:3]: # Show top 3
3452:         print(f"    - {rec['description']} (Priority: {rec['priority']})")
3453:
3454:     # Critical diagnosis summary
3455:     print("\nCRITICAL DIAGNOSIS:")
3456:     diagnosis = results['critical_diagnosis']
3457:     print(f"Critical links identified: {len(diagnosis['critical_links'])}")
3458:
3459:     for link, info in diagnosis['critical_links'].items():
3460:         print(f"\n{link.replace('_', ' ').title()}:")
3461:         print(f"    - Criticality score: {info['criticality_score']:.2f}")
3462:         text_analysis = info['text_analysis']
3463:         print(f"    - Sentiment: {text_analysis['sentiment']}")
3464:         print(f"    - Key words: {' '.join(text_analysis['keywords'][:5])}")
3465:
3466:     # Show risk assessment
3467:     if link in diagnosis['risk_assessment']:
3468:         risk = diagnosis['risk_assessment'][link]
3469:         print(f"    - Risk level: {risk['overall_risk']}")
3470:         if risk['risk_factors']:
3471:             print(f"    - Risk factors: {len(risk['risk_factors'])}")
3472:
3473:     # Show interventions
3474:     if link in diagnosis['intervention_recommendations']:
3475:         interventions = diagnosis['intervention_recommendations'][link]
3476:         print(f"    - Recommended interventions: {len(interventions)}")
3477:
3478:     # Overall summary
3479:     print("\nEXECUTIVE SUMMARY:")
3480:     summary = results['summary']
3481:     print(f"- Document coverage: {summary['document_coverage']['total_segments_analyzed']} segments")
3482:     print(f"- Average efficiency: {summary['performance_summary']['average_efficiency_score']:.2f}")
3483:     print(f"- Overall risk level: {summary['risk_assessment']['overall_risk_level']}")
3484:
3485:     return results
3486:
3487: except FileNotFoundError as e:
3488:     print(f"Error: File not found - {e}")
```

```
3489:         return None
3490:     except Exception as e:
3491:         print(f"Error during analysis: {e}")
3492:         return None
3493:     finally:
3494:         # Clean up
3495:         try:
3496:             import os
3497:             os.remove(SAMPLE_MUNICIPAL_PLAN)
3498:         except (FileNotFoundError, OSError):
3499:             pass
3500:
3501: @dataclass
3502: class CanonicalQuestionContract:
3503:     """Canonical contract linking questionnaire, policy area and evidence."""
3504:
3505:     legacy_question_id: str
3506:     policy_area_id: str
3507:     dimension_id: str
3508:     question_number: int
3509:     expected_elements: list[str]
3510:     search_patterns: dict[str, Any]
3511:     verification_patterns: list[str]
3512:     evaluation_criteria: dict[str, Any]
3513:     question_template: str
3514:     scoring_modality: str
3515:     evidence_sources: dict[str, Any]
3516:     policy_area_legacy: str
3517:     dimension_legacy: str
3518:     canonical_question_id: str = ""
3519:     contract_hash: str = ""
3520:
3521: @dataclass
3522: class EvidenceSegment:
3523:     """Single segment of text matched against a question contract."""
3524:
3525:     segment_index: int
3526:     segment_text: str
3527:     segment_hash: str
3528:     matched_patterns: list[str]
3529:
3530: class CanonicalQuestionSegmenter:
3531:     """Deterministic segmenter anchored to canonical questionnaire schemas."""
3532:
3533:     def __init__(
3534:         self,
3535:         questionnaire_path: str = "questionnaire.json",
3536:         rubric_path: str = "rubric_scoring_FIXED.json",
3537:         segmentation_method: str = "paragraph",
3538:     ) -> None:
3539:         self.questionnaire_path = Path(questionnaire_path)
3540:         self.rubric_path = Path(rubric_path)
3541:         self.segmentation_method = segmentation_method
3542:
3543:     (
3544:         self.contracts,
```

```
3545:         self.questionnaire_metadata,
3546:         self.rubric_metadata,
3547:         self.contracts_hash,
3548:     ) = DocumentProcessor.load_canonical_question_contracts(
3549:         questionnaire_path=questionnaire_path,
3550:         rubric_path=rubric_path,
3551:     )
3552:
3553: @calibrated_method("farfan_core.analysis.Analyzer_one.CanonicalQuestionSegmenter.segment_plan")
3554: def segment_plan(self, plan_text: str) -> dict[str, Any]:
3555:     """Segment *plan_text* and emit evidence manifests per canonical contract."""
3556:
3557:     normalized_text = plan_text or ""
3558:     segments = DocumentProcessor.segment_text(
3559:         normalized_text,
3560:         method=self.segmentation_method,
3561:     )
3562:     normalized_segments = [segment.strip() for segment in segments if segment and segment.strip()]
3563:
3564:     matched_contracts = 0
3565:     question_segments: dict[tuple[str, str, str], dict[str, Any]] = {}
3566:
3567:     for contract in self.contracts:
3568:         manifest = self._build_manifest(contract, normalized_segments)
3569:         if manifest["matched"]:
3570:             matched_contracts += 1
3571:
3572:         key_tuple = (
3573:             contract.canonical_question_id,
3574:             contract.policy_area_id,
3575:             contract.dimension_id,
3576:         )
3577:
3578:         question_segments[key_tuple] = {
3579:             "legacy_question_id": contract.legacy_question_id,
3580:             "policy_area_id": contract.policy_area_id,
3581:             "dimension_id": contract.dimension_id,
3582:             "policy_area_legacy": contract.policy_area_legacy,
3583:             "dimension_legacy": contract.dimension_legacy,
3584:             "question_number": contract.question_number,
3585:             "question_template": contract.question_template,
3586:             "scoring_modality": contract.scoring_modality,
3587:             "evidence_sources": contract.evidence_sources,
3588:             "contract_hash": contract.contract_hash,
3589:             "evidence_manifest": manifest,
3590:         }
3591:
3592:     total_contracts = len(self.contracts)
3593:     metadata = {
3594:         "questionnaire_version": self.questionnaire_metadata.get("version"),
3595:         "rubric_version": self.rubric_metadata.get("version"),
3596:         "total_contracts": total_contracts,
3597:         "covered_contracts": matched_contracts,
3598:         "coverage_ratio": (
3599:             matched_contracts / total_contracts if total_contracts else ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.CanonicalQuestionSegmen
3600: ter.segment_plan", "auto_param_L1127_76", 0.0)
```

```
3600:         ),
3601:         "total_segments": len(normalized_segments),
3602:         "input_sha256": hashlib.sha256(normalized_text.encode("utf-8")).hexdigest(),
3603:         "contracts_sha256": self.contracts_hash,
3604:         "segmentation_method": self.segmentation_method,
3605:     }
3606:
3607:     question_segment_index = [
3608:         {
3609:             "key_tuple": list(key_tuple),
3610:             "canonical_question_id": key_tuple[0],
3611:             "policy_area_id": key_tuple[1],
3612:             "dimension_id": key_tuple[2],
3613:             "legacy_question_id": payload["legacy_question_id"],
3614:             "contract_hash": payload["contract_hash"],
3615:             "evidence_manifest": payload["evidence_manifest"],
3616:         }
3617:         for key_tuple, payload in question_segments.items()
3618:     ]
3619:
3620:     return {
3621:         "metadata": metadata,
3622:         "question_segments": question_segments,
3623:         "question_segment_index": question_segment_index,
3624:     }
3625:
3626: def _build_manifest(
3627:     self,
3628:     contract: CanonicalQuestionContract,
3629:     segments: list[str],
3630: ) -> dict[str, Any]:
3631:     """Build deterministic evidence manifest for *contract* across *segments*."""
3632:
3633:     compiled_patterns: list[tuple[str, Any]] = []
3634:     for element, spec in contract.search_patterns.items():
3635:         pattern = spec.get("pattern") if isinstance(spec, dict) else None
3636:         if not pattern or not isinstance(pattern, str):
3637:             continue
3638:         try:
3639:             compiled_patterns.append(
3640:                 (element, re.compile(pattern, flags=re.IGNORECASE | re.MULTILINE))
3641:             )
3642:         except re.error:
3643:             logger.debug(
3644:                 "Invalid regex pattern skipped",
3645:                 extra={"question_id": contract.legacy_question_id, "pattern": pattern},
3646:             )
3647:
3648:     for index, pattern in enumerate(contract.verification_patterns):
3649:         if not pattern or not isinstance(pattern, str):
3650:             continue
3651:         try:
3652:             compiled_patterns.append(
3653:                 (
3654:                     f"verification_{index}",
3655:                     re.compile(pattern, flags=re.IGNORECASE | re.MULTILINE),
```

```
3656:         )
3657:     )
3658:     except re.error:
3659:         logger.debug(
3660:             "Invalid verification pattern skipped",
3661:             extra={
3662:                 "question_id": contract.legacy_question_id,
3663:                 "pattern_index": index,
3664:             },
3665:         )
3666:
3667: matched_segments: list[EvidenceSegment] = []
3668: pattern_hits: dict[str, int] = {}
3669:
3670: for segment_index, segment_text in enumerate(segments):
3671:     matched_labels: list[str] = []
3672:     for label, pattern in compiled_patterns:
3673:         if pattern.search(segment_text):
3674:             matched_labels.append(label)
3675:
3676:     if matched_labels:
3677:         unique_labels = sorted(set(matched_labels))
3678:         segment_hash = hashlib.sha256(segment_text.encode("utf-8")).hexdigest()
3679:         matched_segments.append(
3680:             EvidenceSegment(
3681:                 segment_index=segment_index,
3682:                 segment_text=segment_text,
3683:                 segment_hash=segment_hash,
3684:                 matched_patterns=unique_labels,
3685:             )
3686:         )
3687:
3688:         for label in unique_labels:
3689:             pattern_hits[label] = pattern_hits.get(label, 0) + 1
3690:
3691: manifest_segments = [
3692:     {
3693:         "segment_index": segment.segment_index,
3694:         "segment_text": segment.segment_text,
3695:         "segment_hash": segment.segment_hash,
3696:         "matched_patterns": segment.matched_patterns,
3697:     }
3698:     for segment in matched_segments
3699: ]
3700:
3701: segment_hash_chain = (
3702:     hashlib.sha256(
3703:         "".join(segment["segment_hash"] for segment in manifest_segments).encode("utf-8")
3704:     ).hexdigest()
3705:     if manifest_segments
3706:     else "0" * 64
3707: )
3708:
3709: return {
3710:     "matched": bool(manifest_segments),
3711:     "matched_segment_count": len(manifest_segments),
```

```
3712:         "expected_elements": contract.expected_elements,
3713:         "search_patterns": contract.search_patterns,
3714:         "verification_patterns": contract.verification_patterns,
3715:         "evaluation_criteria": contract.evaluation_criteria,
3716:         "pattern_hits": pattern_hits,
3717:         "matched_segments": manifest_segments,
3718:         "attestation": {
3719:             "contract_sha256": contract.contract_hash,
3720:             "segment_hash_chain": segment_hash_chain,
3721:         },
3722:     }
3723:
3724: class DocumentProcessor:
3725:     """Utility class for document processing."""
3726:
3727:     @staticmethod
3728:     def load_pdf(pdf_path: str) -> str:
3729:         """Load text from PDF file."""
3730:         try:
3731:             # Delegate to factory for I/O operation
3732:             # Note: PyPDF2 requires file handle, so we need a special approach
3733:             from pathlib import Path
3734:
3735:             import PyPDF2
3736:             pdf_path_obj = Path(pdf_path)
3737:
3738:             with open(pdf_path_obj, 'rb') as file:
3739:                 reader = PyPDF2.PdfReader(file)
3740:                 text = ""
3741:                 for page in reader.pages:
3742:                     text += page.extract_text()
3743:                 return text
3744:         except ImportError:
3745:             logger.warning("PyPDF2 not available. Install with: pip install PyPDF2")
3746:             return ""
3747:         except Exception as e:
3748:             logger.error(f"Error loading PDF: {e}")
3749:             return ""
3750:
3751:     @staticmethod
3752:     def load_docx(docx_path: str) -> str:
3753:         """Load text from DOCX file."""
3754:         try:
3755:             import docx
3756:             doc = docx.Document(docx_path)
3757:             text = ""
3758:             for paragraph in doc.paragraphs:
3759:                 text += paragraph.text + "\n"
3760:             return text
3761:         except ImportError:
3762:             logger.warning("python-docx not available. Install with: pip install python-docx")
3763:             return ""
3764:         except Exception as e:
3765:             logger.error(f"Error loading DOCX: {e}")
3766:             return ""
3767:
```

```
3768: @staticmethod
3769: def segment_text(text: str, method: str = "sentence") -> list[str]:
3770:     """Segment text using different methods."""
3771:
3772:     if method == "sentence":
3773:         # Use NLTK sentence tokenizer if available
3774:         if sent_tokenize is not None:
3775:             try:
3776:                 return sent_tokenize(text, language='spanish')
3777:             except LookupError:
3778:                 # Download if not available
3779:                 try:
3780:                     import nltk
3781:                     nltk.download('punkt')
3782:                     return sent_tokenize(text, language='spanish')
3783:                 except:
3784:                     # Fallback to simple splitting
3785:                     return [s.strip() for s in re.split(r'[.!?]+', text) if len(s.strip()) > 10]
3786:             except Exception:
3787:                 # Fallback to simple splitting
3788:                 return [s.strip() for s in re.split(r'[.!?]+', text) if len(s.strip()) > 10]
3789:         else:
3790:             # Fallback to simple splitting
3791:             return [s.strip() for s in re.split(r'[.!?]+', text) if len(s.strip()) > 10]
3792:
3793:     elif method == "paragraph":
3794:         return [p.strip() for p in text.split('\n\n') if len(p.strip()) > 20]
3795:
3796:     elif method == "fixed_length":
3797:         words = text.split()
3798:         segments = []
3799:         segment_length = 50 # words per segment
3800:
3801:         for i in range(0, len(words), segment_length):
3802:             segment = " ".join(words[i:i + segment_length])
3803:             if len(segment) > 20:
3804:                 segments.append(segment)
3805:
3806:         return segments
3807:
3808:     else:
3809:         raise ValueError(f"Unknown segmentation method: {method}")
3810:
3811: @staticmethod
3812: def load_canonical_question_contracts(
3813:     questionnaire_path: str = "questionnaire.json",
3814:     rubric_path: str = "rubric_scoring_FIXED.json",
3815: ) -> tuple[list[CanonicalQuestionContract], dict[str, Any], dict[str, Any], str]:
3816:     """Load canonical question contracts based on questionnaire and rubric."""
3817:
3818:     questionnaire_file = Path(questionnaire_path)
3819:     rubric_file = Path(rubric_path)
3820:
3821:     if not questionnaire_file.exists():
3822:         raise FileNotFoundError(f"Questionnaire file not found: {questionnaire_file}")
3823:     if not rubric_file.exists():
```

```
3824:         raise FileNotFoundError(f"Rubric file not found: {rubric_file}")
3825:
3826:     # Delegate to factory for I/O operation
3827:     from farfan_pipeline.analysis.factory import load_json
3828:
3829:     questionnaire_data = load_json(questionnaire_file)
3830:     rubric_data = load_json(rubric_file)
3831:
3832:     questionnaire_meta = questionnaire_data.get("metadata", {})
3833:     rubric_meta = rubric_data.get("metadata", {})
3834:
3835:     policy_area_mapping = questionnaire_meta.get("policy_area_mapping", {})
3836:     inverse_policy_area_map = {
3837:         legacy: canonical
3838:         for canonical, legacy in policy_area_mapping.items()
3839:         if isinstance(legacy, str)
3840:     }
3841:
3842:     base_questions = questionnaire_data.get("preguntas_base", [])
3843:     questionnaire_lookup: dict[tuple[str, str, int], dict[str, Any]] = {}
3844:     for question in base_questions:
3845:         if not isinstance(question, dict):
3846:             continue
3847:         legacy_question_id = question.get("id")
3848:         if not legacy_question_id:
3849:             continue
3850:         legacy_policy_area = (
3851:             question.get("metadata", {}).get("policy_area")
3852:             or legacy_question_id.split("-")[0]
3853:         )
3854:         dimension_legacy = question.get("dimension") or legacy_question_id.split("-")[1]
3855:         try:
3856:             question_number = int(str(question.get("numero")))
3857:         except (TypeError, ValueError):
3858:             question_number = 0
3859:         key = (legacy_policy_area, dimension_legacy, question_number)
3860:         questionnaire_lookup[key] = question
3861:
3862:     rubric_questions = rubric_data.get("questions", [])
3863:     rubric_lookup: dict[tuple[str, str, int], dict[str, Any]] = {}
3864:     for question in rubric_questions:
3865:         if not isinstance(question, dict):
3866:             continue
3867:         legacy_question_id = question.get("id")
3868:         if not legacy_question_id:
3869:             continue
3870:         legacy_policy_area = question.get("policy_area") or legacy_question_id.split("-")[0]
3871:         dimension_legacy = question.get("dimension") or legacy_question_id.split("-")[1]
3872:         try:
3873:             raw_number = int(str(question.get("question_no")))
3874:         except (TypeError, ValueError):
3875:             raw_number = 0
3876:         normalized_number = ((raw_number - 1) % 5) + 1 if raw_number else 0
3877:         key = (legacy_policy_area, dimension_legacy, normalized_number)
3878:         rubric_lookup[key] = question
3879:
```

```
3880: common_keys = sorted(set(questionnaire_lookup.keys()) & set(rubric_lookup.keys()))
3881: if not common_keys:
3882:     raise ValueError("No overlapping question definitions between questionnaire and rubric metadata")
3883:
3884: contracts: list[CanonicalQuestionContract] = []
3885:
3886: for key in common_keys:
3887:     questionnaire_entry = questionnaire_lookup[key]
3888:     rubric_entry = rubric_lookup[key]
3889:     legacy_question_id = questionnaire_entry.get("id") or rubric_entry.get("id")
3890:
3891:     legacy_policy_area = (
3892:         questionnaire_entry.get("metadata", {}).get("policy_area")
3893:         or rubric_entry.get("policy_area", "")
3894:     )
3895:     canonical_policy_area = inverse_policy_area_map.get(
3896:         legacy_policy_area,
3897:         DocumentProcessor._default_policy_area_id(legacy_policy_area),
3898:     )
3899:
3900:     dimension_legacy = (
3901:         questionnaire_entry.get("dimension")
3902:         or rubric_entry.get("dimension", "")
3903:     )
3904:     canonical_dimension = DocumentProcessor._to_canonical_dimension_id(dimension_legacy)
3905:
3906:     question_number_value = (
3907:         rubric_entry.get("question_no")
3908:         if rubric_entry.get("question_no") is not None
3909:         else questionnaire_entry.get("numero")
3910:     )
3911:     try:
3912:         question_number = int(str(question_number_value).lstrip("Qq"))
3913:     except (TypeError, ValueError):
3914:         question_number = 0
3915:
3916:     expected_elements = rubric_entry.get("expected_elements", [])
3917:     if not isinstance(expected_elements, list):
3918:         expected_elements = []
3919:
3920:     search_patterns = rubric_entry.get("search_patterns", {})
3921:     if not isinstance(search_patterns, dict):
3922:         search_patterns = {}
3923:
3924:     verification_patterns = questionnaire_entry.get("patrones_verificacion", [])
3925:     if not isinstance(verification_patterns, list):
3926:         verification_patterns = []
3927:
3928:     evaluation_criteria = questionnaire_entry.get("criterios_evaluacion", {})
3929:     if not isinstance(evaluation_criteria, dict):
3930:         evaluation_criteria = {}
3931:
3932:     evidence_sources = rubric_entry.get("evidence_sources", {})
3933:     if not isinstance(evidence_sources, dict):
3934:         evidence_sources = {}
3935:
```

```
3936:         contract = CanonicalQuestionContract(
3937:             legacy_question_id=legacy_question_id,
3938:             policy_area_id=canonical_policy_area,
3939:             dimension_id=canonical_dimension,
3940:             question_number=question_number,
3941:             expected_elements=expected_elements,
3942:             search_patterns=search_patterns,
3943:             verification_patterns=verification_patterns,
3944:             evaluation_criteria=evaluation_criteria,
3945:             question_template=(
3946:                 rubric_entry.get("template")
3947:                 or questionnaire_entry.get("texto_template", "")
3948:             ),
3949:             scoring_modality=rubric_entry.get("scoring_modality", ""),
3950:             evidence_sources=evidence_sources,
3951:             policy_area_legacy=legacy_policy_area,
3952:             dimension_legacy=dimension_legacy,
3953:         )
3954:
3955:         contracts.append(contract)
3956:
3957:     contracts.sort(
3958:         key=lambda contract: (
3959:             contract.policy_area_id,
3960:             contract.dimension_id,
3961:             contract.question_number,
3962:             contract.legacy_question_id,
3963:         )
3964:     )
3965:
3966:     for index, contract in enumerate(contracts, start=1):
3967:         canonical_question_id = f"Q{index:03d}"
3968:         contract.canonical_question_id = canonical_question_id
3969:         payload = {
3970:             "canonical_question_id": canonical_question_id,
3971:             "legacy_question_id": contract.legacy_question_id,
3972:             "policy_area_id": contract.policy_area_id,
3973:             "dimension_id": contract.dimension_id,
3974:             "question_number": contract.question_number,
3975:             "expected_elements": contract.expected_elements,
3976:             "search_patterns": contract.search_patterns,
3977:             "verification_patterns": contract.verification_patterns,
3978:             "evaluation_criteria": contract.evaluation_criteria,
3979:             "question_template": contract.question_template,
3980:             "scoring_modality": contract.scoring_modality,
3981:             "evidence_sources": contract.evidence_sources,
3982:         }
3983:         contract.contract_hash = hashlib.sha256(
3984:             json.dumps(payload, sort_keys=True, ensure_ascii=False).encode("utf-8")
3985:         ).hexdigest()
3986:
3987:     contracts_hash = (
3988:         hashlib.sha256(
3989:             "".join(contract.contract_hash for contract in contracts).encode("utf-8")
3990:         ).hexdigest()
3991:         if contracts
```

```
3992:         else "0" * 64
3993:     )
3994:
3995:     return contracts, questionnaire_meta, rubric_meta, contracts_hash
3996:
3997: @staticmethod
3998: def segment_by_canonical_questionnaire(
3999:     plan_text: str,
4000:     questionnaire_path: str = "questionnaire.json",
4001:     rubric_path: str = "rubric_scoring_FIXED.json",
4002:     segmentation_method: str = "paragraph",
4003: ) -> dict[str, Any]:
4004:     """Convenience wrapper to segment plan text using canonical contracts."""
4005:
4006:     segmenter = CanonicalQuestionSegmenter(
4007:         questionnaire_path=questionnaire_path,
4008:         rubric_path=rubric_path,
4009:         segmentation_method=segmentation_method,
4010:     )
4011:     return segmenter.segment_plan(plan_text)
4012:
4013: @staticmethod
4014: def _default_policy_area_id(legacy_policy_area: str) -> str:
4015:     """Convert legacy policy-area code (e.g., P1) into canonical PAxx format."""
4016:
4017:     if isinstance(legacy_policy_area, str) and legacy_policy_area.startswith("P"):
4018:         try:
4019:             return f"PA{int(legacy_policy_area[1:]):02d}"
4020:         except ValueError:
4021:             return legacy_policy_area
4022:     return legacy_policy_area
4023:
4024: @staticmethod
4025: def _to_canonical_dimension_id(dimension_code: str) -> str:
4026:     """Convert legacy dimension code (e.g., D1) into canonical DIMxx format."""
4027:
4028:     if isinstance(dimension_code, str) and dimension_code.startswith("D"):
4029:         try:
4030:             return f"DIM{int(dimension_code[1:]):02d}"
4031:         except ValueError:
4032:             return dimension_code
4033:     return dimension_code
4034:
4035: class ResultsExporter:
4036:     """Export analysis results to different formats."""
4037:
4038:     @staticmethod
4039:     def export_to_json(results: dict[str, Any], output_path: str) -> None:
4040:         """Export results to JSON file."""
4041:         # Delegate to factory for I/O operation
4042:         from farfan_pipeline.analysis.factory import save_json
4043:
4044:         try:
4045:             save_json(results, output_path)
4046:             logger.info(f"Results exported to JSON: {output_path}")
4047:         except Exception as e:
```

```
4048:         logger.error(f"Error exporting to JSON: {e}")
4049:
4050:     @staticmethod
4051:     def export_to_excel(results: dict[str, Any], output_path: str) -> None:
4052:         """Export results to Excel file."""
4053:         if pd is None:
4054:             logger.warning("pandas not available. Install with: pip install pandas openpyxl")
4055:             return
4056:
4057:         try:
4058:             with pd.ExcelWriter(output_path, engine='openpyxl') as writer:
4059:
4060:                 # Summary sheet
4061:                 summary_data = []
4062:                 summary = results.get('summary', {})
4063:
4064:                 for category, data in summary.items():
4065:                     if isinstance(data, dict):
4066:                         for key, value in data.items():
4067:                             summary_data.append({
4068:                                 'Category': category,
4069:                                 'Metric': key,
4070:                                 'Value': value
4071:                             })
4072:
4073:                 if summary_data:
4074:                     pd.DataFrame(summary_data).to_excel(writer, sheet_name='Summary', index=False)
4075:
4076:                 # Performance metrics sheet
4077:                 perf_data = []
4078:                 perf_analysis = results.get('performance_analysis', {})
4079:
4080:                 for link, metrics in perf_analysis.get('value_chain_metrics', {}).items():
4081:                     perf_data.append({
4082:                         'Value_Chain_Link': link,
4083:                         'Efficiency_Score': metrics.get('efficiency_score', 0),
4084:                         'Throughput': metrics.get('throughput', 0),
4085:                         'Capacity_Utilization': metrics.get('capacity_utilization', 0),
4086:                         'Segment_Count': metrics.get('segment_count', 0)
4087:                     })
4088:
4089:                 if perf_data:
4090:                     pd.DataFrame(perf_data).to_excel(writer, sheet_name='Performance', index=False)
4091:
4092:                 # Recommendations sheet
4093:                 rec_data = []
4094:                 recommendations = perf_analysis.get('optimization_recommendations', [])
4095:
4096:                 for i, rec in enumerate(recommendations):
4097:                     rec_data.append({
4098:                         'Recommendation_ID': i + 1,
4099:                         'Link': rec.get('link', ''),
4100:                         'Type': rec.get('type', ''),
4101:                         'Priority': rec.get('priority', ''),
4102:                         'Description': rec.get('description', '')
4103:                     })
```

```
4104:
4105:         if rec_data:
4106:             pd.DataFrame(rec_data).to_excel(writer, sheet_name='Recommendations', index=False)
4107:
4108:         logger.info(f"Results exported to Excel: {output_path}")
4109:
4110:     except ImportError:
4111:         logger.warning("openpyxl not available. Install with: pip install openpyxl")
4112:     except Exception as e:
4113:         logger.error(f"Error exporting to Excel: {e}")
4114:
4115: @staticmethod
4116: def export_summary_report(results: dict[str, Any], output_path: str) -> None:
4117:     """Export a summary report in text format."""
4118:
4119:     try:
4120:         # Build content first
4121:         lines = []
4122:         lines.append("MUNICIPAL DEVELOPMENT PLAN ANALYSIS REPORT\n")
4123:         lines.append("=" * 50 + "\n\n")
4124:
4125:         # Basic info
4126:         lines.append(f"Document: {results.get('document_path', 'Unknown')}\n")
4127:         lines.append(f"Analysis Date: {results.get('analysis_timestamp', 'Unknown')}\n")
4128:         lines.append(f"Processing Time: {results.get('processing_time_seconds', 0):.2f} seconds\n\n")
4129:
4130:         # Summary
4131:         summary = results.get('summary', {})
4132:         lines.append("EXECUTIVE SUMMARY\n")
4133:         lines.append("-" * 20 + "\n")
4134:
4135:         doc_coverage = summary.get('document_coverage', {})
4136:         lines.append(f"Segments Analyzed: {doc_coverage.get('total_segments_analyzed', 0)}\n")
4137:         lines.append(f"Value Chain Links: {doc_coverage.get('value_chain_links_identified', 0)}\n")
4138:         lines.append(f"Policy Domains: {doc_coverage.get('policy_domains_covered', 0)}\n")
4139:
4140:         perf_summary = summary.get('performance_summary', {})
4141:         lines.append(f"Average Efficiency: {perf_summary.get('average_efficiency_score', 0):.2f}\n")
4142:
4143:         risk_summary = summary.get('risk_assessment', {})
4144:         lines.append(f"Overall Risk Level: {risk_summary.get('overall_risk_level', 'Unknown')}\n\n")
4145:
4146:         # Performance details
4147:         lines.append("PERFORMANCE ANALYSIS\n")
4148:         lines.append("-" * 20 + "\n")
4149:
4150:         perf_analysis = results.get('performance_analysis', {})
4151:         for link, metrics in perf_analysis.get('value_chain_metrics', {}).items():
4152:             lines.append(f"\n{link.replace('_', ' ').title()}\n")
4153:             lines.append(f"    Efficiency: {metrics.get('efficiency_score', 0):.2f}\n")
4154:             lines.append(f"    Throughput: {metrics.get('throughput', 0):.1f}\n")
4155:             lines.append(f"    Capacity: {metrics.get('capacity_utilization', 0):.2f}\n")
4156:
4157:         # Recommendations
4158:         lines.append("\n\nRECOMMENDATION OPTIONS\n")
4159:         lines.append("-" * 20 + "\n")
```

```
4160:
4161:     recommendations = perf_analysis.get('optimization_recommendations', [])
4162:     for i, rec in enumerate(recommendations, 1):
4163:         lines.append(f"{i}. {rec.get('description', '')} (Priority: {rec.get('priority', '')})\n")
4164:
4165:     # Critical links
4166:     lines.append("\n\nCRITICAL LINKS\n")
4167:     lines.append("-" * 15 + "\n")
4168:
4169:     diagnosis = results.get('critical_diagnosis', {})
4170:     for link, info in diagnosis.get('critical_links', {}).items():
4171:         lines.append(f"\n{link.replace('_', ' ').title()}\n")
4172:         lines.append(f"    Criticality: {info.get('criticality_score', 0):.2f}\n")
4173:
4174:     text_analysis = info.get('text_analysis', {})
4175:     lines.append(f"    Sentiment: {text_analysis.get('sentiment', 'neutral')}\n")
4176:
4177:     if link in diagnosis.get('risk_assessment', {}):
4178:         risk = diagnosis['risk_assessment'][link]
4179:         lines.append(f"    Risk Level: {risk.get('overall_risk', 'unknown')}\n")
4180:
4181:     # Delegate to factory for I/O operation
4182:     from farfan_pipeline.analysis.factory import write_text_file
4183:     write_text_file(''.join(lines), output_path)
4184:     logger.info(f"Summary report exported: {output_path}")
4185:
4186: except Exception as e:
4187:     logger.error(f"Error exporting summary report: {e}")
4188:
4189: # -----
4190: # 7. MAIN EXECUTION
4191: # -----
4192:
4193: # -----
4194: # 8. ADDITIONAL UTILITIES FOR PRODUCTION USE
4195: # -----
4196:
4197: class ConfigurationManager:
4198:     """Manage analyzer configuration."""
4199:
4200:     def __init__(self, config_path: str | None = None) -> None:
4201:         self.config_path = config_path or "analyzer_config.json"
4202:         self.config = self.load_config()
4203:
4204:     @calibrated_method("farfan_core.analysis.Analyzer_one.ConfigurationManager.load_config")
4205:     def load_config(self) -> dict[str, Any]:
4206:         """Load configuration from file or create default."""
4207:
4208:         default_config = {
4209:             "processing": {
4210:                 "max_segments": 200,
4211:                 "min_segment_length": 20,
4212:                 "segmentation_method": "sentence"
4213:             },
4214:             "analysis": {
4215:                 "criticality_threshold": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.ConfigurationManager.load_config", "auto_param_L1743_41",
```

```
0.4),
4216:         "efficiency_threshold": ParameterLoaderV2.get("farfan_core.analysis.Analyzer_one.ConfigurationManager.load_config", "auto_param_L1744_40", 0
.5),
4217:         "throughput_threshold": 20
4218:     },
4219:     "export": {
4220:         "include_raw_data": False,
4221:         "export_formats": ["json", "excel", "summary"]
4222:     }
4223: }
4224:
4225: if Path(self.config_path).exists():
4226:     # Delegate to factory for I/O operation
4227:     from farfan_pipeline.analysis.factory import load_json
4228:
4229:     try:
4230:         user_config = load_json(self.config_path)
4231:         # Merge with defaults
4232:         for key, value in user_config.items():
4233:             if key in default_config and isinstance(value, dict):
4234:                 default_config[key].update(value)
4235:             else:
4236:                 default_config[key] = value
4237:     except Exception as e:
4238:         logger.warning(f"Error loading config: {e}. Using defaults.")
4239:
4240:     return default_config
4241:
4242: @calibrated_method("farfan_core.analysis.Analyzer_one.ConfigurationManager.save_config")
4243: def save_config(self) -> None:
4244:     """Save current configuration to file."""
4245:     # Delegate to factory for I/O operation
4246:     from farfan_pipeline.analysis.factory import save_json
4247:
4248:     try:
4249:         save_json(self.config, self.config_path)
4250:     except Exception as e:
4251:         logger.error(f"Error saving config: {e}")
4252:
4253: class BatchProcessor:
4254:     """Process multiple documents in batch."""
4255:
4256:     def __init__(self, analyzer: MunicipalAnalyzer) -> None:
4257:         self.analyzer = analyzer
4258:
4259:     @calibrated_method("farfan_core.analysis.Analyzer_one.BatchProcessor.process_directory")
4260:     def process_directory(self, directory_path: str, pattern: str = "*.txt") -> dict[str, Any]:
4261:         """Process all files matching pattern in directory."""
4262:
4263:         directory = Path(directory_path)
4264:         if not directory.exists():
4265:             raise ValueError(f"Directory not found: {directory_path}")
4266:
4267:         files = list(directory.glob(pattern))
4268:         results = {}
4269:
```

```
4270:         logger.info(f"Processing {len(files)} files from {directory_path}")
4271:
4272:     for file_path in files:
4273:         try:
4274:             logger.info(f"Processing: {file_path.name}")
4275:             result = self.analyzer.analyze_document(str(file_path))
4276:             results[file_path.name] = result
4277:         except Exception as e:
4278:             logger.error(f"Error processing {file_path.name}: {e}")
4279:             results[file_path.name] = {"error": str(e)}
4280:
4281:     return results
4282:
4283: @calibrated_method("farfan_core.analysis.Analyzer_one.BatchProcessor.export_batch_results")
4284: def export_batch_results(self, batch_results: dict[str, Any], output_dir: str) -> None:
4285:     """Export batch processing results."""
4286:
4287:     output_path = Path(output_dir)
4288:     output_path.mkdir(exist_ok=True)
4289:
4290:     # Export individual results
4291:     for filename, result in batch_results.items():
4292:         if "error" not in result:
4293:             base_name = Path(filename).stem
4294:
4295:             # JSON export
4296:             json_path = output_path / f"{base_name}_results.json"
4297:             ResultsExporter.export_to_json(result, str(json_path))
4298:
4299:             # Summary export
4300:             summary_path = output_path / f"{base_name}_summary.txt"
4301:             ResultsExporter.export_summary_report(result, str(summary_path))
4302:
4303:     # Create batch summary
4304:     self._create_batch_summary(batch_results, output_path)
4305:
4306: @calibrated_method("farfan_core.analysis.Analyzer_one.BatchProcessor._create_batch_summary")
4307: def _create_batch_summary(self, batch_results: dict[str, Any], output_path: Path) -> None:
4308:     """Create summary of batch processing results."""
4309:
4310:     summary_file = output_path / "batch_summary.txt"
4311:
4312:     try:
4313:         # Build content first
4314:         lines = []
4315:         lines.append("BATCH PROCESSING SUMMARY\n")
4316:         lines.append("=" * 30 + "\n\n")
4317:
4318:         total_files = len(batch_results)
4319:         successful = sum(1 for r in batch_results.values() if "error" not in r)
4320:         failed = total_files - successful
4321:
4322:         lines.append(f"Total files processed: {total_files}\n")
4323:         lines.append(f"Successful: {successful}\n")
4324:         lines.append(f"Failed: {failed}\n\n")
4325:
```

```
4326:         if failed > 0:
4327:             lines.append("FAILED FILES:\n")
4328:             lines.append("-" * 15 + "\n")
4329:             for filename, result in batch_results.items():
4330:                 if "error" in result:
4331:                     lines.append(f"- {filename}: {result['error']}\n")
4332:             lines.append("\n")
4333:
4334:         if successful > 0:
4335:             lines.append("SUCCESSFUL ANALYSES:\n")
4336:             lines.append("-" * 20 + "\n")
4337:
4338:             for filename, result in batch_results.items():
4339:                 if "error" not in result:
4340:                     summary = result.get('summary', {})
4341:                     perf_summary = summary.get('performance_summary', {})
4342:                     risk_summary = summary.get('risk_assessment', {})
4343:
4344:                     lines.append(f"\n{filename}:\n")
4345:                     lines.append(f"    Efficiency: {perf_summary.get('average_efficiency_score', 0):.2f}\n")
4346:                     lines.append(f"    Risk Level: {risk_summary.get('overall_risk_level', 'unknown')}\n")
4347:
4348:             # Delegate to factory for I/O operation
4349:             from farfan_pipeline.analysis.factory import write_text_file
4350:             write_text_file(''.join(lines), summary_file)
4351:             logger.info(f"Batch summary created: {summary_file}")
4352:
4353:         except Exception as e:
4354:             logger.error(f"Error creating batch summary: {e}")
4355:
4356: # Simple CLI interface
4357: def main() -> None:
4358:     """Simple command-line interface."""
4359:     import argparse
4360:
4361:     parser = argparse.ArgumentParser(description="Municipal Development Plan Analyzer")
4362:     parser.add_argument("input", help="Input file or directory path")
4363:     parser.add_argument("--output", "-o", default=".", help="Output directory")
4364:     parser.add_argument("--batch", "-b", action="store_true", help="Batch process directory")
4365:     parser.add_argument("--config", "-c", help="Configuration file path")
4366:
4367:     args = parser.parse_args()
4368:
4369:     # Initialize analyzer
4370:     analyzer = MunicipalAnalyzer()
4371:
4372:     if args.batch:
4373:         # Batch processing
4374:         processor = BatchProcessor(analyzer)
4375:         results = processor.process_directory(args.input)
4376:         processor.export_batch_results(results, args.output)
4377:         print(f"Batch processing complete. Results in: {args.output}")
4378:     else:
4379:         # Single file processing
4380:         results = analyzer.analyze_document(args.input)
4381:
```

```
4382: # Export results
4383: exporter = ResultsExporter()
4384: output_base = Path(args.output) / Path(args.input).stem
4385:
4386: exporter.export_to_json(results, f"{output_base}_results.json")
4387: exporter.export_summary_report(results, f"{output_base}_summary.txt")
4388:
4389: print(f"Analysis complete. Results in: {args.output}")
4390:
4391:
4392:
4393: =====
4394: FILE: src/farfan_pipeline/analysis/__init__.py
4395: =====
4396:
4397: """Analysis modules for semantic and structural analysis.
4398:
4399: This module exposes production-ready utilities and meso-level analytics functions
4400: for policy analysis workflows.
4401:
4402: Production-Ready Utilities:
4403: -----
4404: - :func:`compute_graph_metrics_with_fallback`: Compute graph metrics with NetworkX
4405:   fallback handling and observability integration
4406: - :func:`compute_basic_graph_stats`: Compute basic graph statistics without NetworkX
4407: - :func:`check_networkx_available`: Check if NetworkX is available for graph computation
4408: - :class:`RetryHandler`: Robust retry mechanism for transient dependency failures
4409: - :class:`SPCCausalBridge`: Convert SPC chunk graphs to causal DAG representations
4410:
4411: Meso-Level Analytics:
4412: -----
4413: - :func:`analyze_policy_dispersion`: Evaluate intra-cluster dispersion with CV, gap
4414:   analysis, and light penalty framework
4415: - :func:`reconcile_cross_metrics`: Validate heterogeneous metric feeds against
4416:   authoritative macro reference with governance flags
4417: - :func:`compose_cluster_posterior`: Aggregate micro posteriors using Bayesian-style
4418:   roll-up with reconciliation penalties
4419: - :func:`calibrate_against_peers`: Situate cluster against peer group using
4420:   inter-quartile comparisons and Tukey-style outlier detection
4421:
4422: Experimental Components:
4423: -----
4424: The following classes are under active development and should be considered
4425: experimental. They are not exposed in the public API but remain available
4426: for research and development purposes:
4427:
4428: - :class:`BayesianMultilevelSystem`: Multi-level Bayesian analysis framework
4429:   (from bayesian_multilevel_system.py)
4430: - :class:`ContradictionDetector`: Transformer-based contradiction detection
4431:   (from contradiction_deteccion.py)
4432: - :class:`DerekBeachCausalFramework`: Process tracing causal analysis
4433:   (from derek_beach.py)
4434: - :class:`TeoriaCambio`: Theory of change validation framework
4435:   (from teoria_cambio.py)
4436:
4437: These experimental components may have unstable APIs, incomplete documentation,
```

```
4438: or require additional dependencies not enforced by the core system.
4439: """
4440:
4441: from farfan_pipeline.analysis.graph_metrics_fallback import (
4442:     check_networkx_available,
4443:     compute_basic_graph_stats,
4444:     compute_graph_metrics_with_fallback,
4445: )
4446: from farfan_pipeline.analysis.meso_cluster_analysis import (
4447:     analyze_policy_dispersion,
4448:     calibrate_against_peers,
4449:     compose_cluster_posterior,
4450:     reconcile_cross_metrics,
4451: )
4452: from farfan_pipeline.analysis.retry_handler import RetryHandler
4453: from farfan_pipeline.analysis.spc_causal_bridge import SPCCausalBridge
4454:
4455: __all__ = [
4456:     "compute_graph_metrics_with_fallback",
4457:     "compute_basic_graph_stats",
4458:     "check_networkx_available",
4459:     "RetryHandler",
4460:     "SPCCausalBridge",
4461:     "analyze_policy_dispersion",
4462:     "reconcile_cross_metrics",
4463:     "compose_cluster_posterior",
4464:     "calibrate_against_peers",
4465: ]
4466:
4467:
4468:
4469: =====
4470: FILE: src/farfan_pipeline/analysis/bayesian_multilevel_system.py
4471: =====
4472:
4473: """
4474: Bayesian Multi-Level Analysis System
4475: =====
4476:
4477: Complete implementation of the multi-level Bayesian analysis framework with:
4478:
4479: MICRO LEVEL:
4480: - Reconciliation Layer: Range/unit/period/entity validators with penalty factors
4481: - Bayesian Updater: Probative test taxonomy with posterior estimation
4482: - Output: posterior_table_micro.csv
4483:
4484: MESO LEVEL:
4485: - Dispersion Engine: CV, max_gap, Gini coefficient computation
4486: - Peer Calibration: peer_context comparison with narrative hooks
4487: - Bayesian Roll-Up: posterior_meso calculation with penalties
4488: - Output: posterior_table_meso.csv
4489:
4490: MACRO LEVEL:
4491: - Contradiction Scanner: micro\206\224meso\206\224macro consistency detector
4492: - Bayesian Portfolio Composer: Coverage, dispersion, contradiction penalties
4493: - Output: posterior_table_macro.csv
```

```
4494:
4495: Author: Integration Team
4496: Version: 1.0.0
4497: Python: 3.10+
4498: """
4499:
4500: from __future__ import annotations
4501:
4502: import logging
4503: from dataclasses import dataclass, field
4504: from enum import Enum, auto
4505: from pathlib import Path
4506: from typing import Any
4507:
4508: import numpy as np
4509: from scipy import stats
4510: from farfan_pipeline.core.parameters import ParameterLoaderV2
4511: from farfan_pipeline.core.calibration.decorators import calibrated_method
4512:
4513: # Configure logging
4514: logging.basicConfig(
4515:     level=logging.INFO,
4516:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
4517: )
4518: logger = logging.getLogger(__name__)
4519:
4520: # =====
4521: # ENUMERATIONS AND TYPE DEFINITIONS
4522: # =====
4523:
4524: class ValidatorType(Enum):
4525:     """Types of validators for reconciliation layer"""
4526:     RANGE = auto()
4527:     UNIT = auto()
4528:     PERIOD = auto()
4529:     ENTITY = auto()
4530:
4531: class ProbativeTestType(Enum):
4532:     """Taxonomy of probative tests for Bayesian updating"""
4533:     STRAW_IN_WIND = "straw_in_wind" # Weak confirmation
4534:     HOOP_TEST = "hoop_test" # Necessary but not sufficient
4535:     SMOKING_GUN = "smoking_gun" # Sufficient but not necessary
4536:     DOUBLY_DECISIVE = "doubly_decisive" # Both necessary and sufficient
4537:
4538: class PenaltyCategory(Enum):
4539:     """Categories of penalties applied to scores"""
4540:     VALIDATION_FAILURE = "validation_failure"
4541:     DISPERSION_HIGH = "dispersion_high"
4542:     COVERAGE_GAP = "coverage_gap"
4543:     CONTRADICTION = "contradiction"
4544:     PEER_DEVIATION = "peer_deviation"
4545:
4546: # =====
4547: # MICRO LEVEL: RECONCILIATION LAYER
4548: # =====
4549:
```

```

4550: @dataclass
4551: class ValidationRule:
4552:     """Definition of a validation rule"""
4553:     validator_type: ValidatorType
4554:     field_name: str
4555:     expected_range: tuple[float, float] | None = None
4556:     expected_unit: str | None = None
4557:     expected_period: str | None = None
4558:     expected_entity: str | None = None
4559:     penalty_factor: float = 0.1 # Penalty multiplier for violations
4560:
4561: @dataclass
4562: class ValidationResult:
4563:     """Result of a validation check"""
4564:     rule: ValidationRule
4565:     passed: bool
4566:     observed_value: Any
4567:     expected_value: Any
4568:     violation_severity: float # 0.0 (no violation) to 1.0 (severe)
4569:     penalty_applied: float
4570:
4571: class ReconciliationValidator:
4572:     """
4573:     Reconciliation Layer: Validates data against expected ranges, units, periods, entities
4574:     Applies penalty factors for violations
4575:     """
4576:
4577:     def __init__(self, validation_rules: list[ValidationRule]) -> None:
4578:         self.rules = validation_rules
4579:         self.logger = logging.getLogger(self.__class__.__name__)
4580:
4581:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range")
4582:     def validate_range(self, value: float, rule: ValidationRule) -> ValidationResult:
4583:         """Validate numeric value is within expected range"""
4584:         if rule.expected_range is None:
4585:             return ValidationResult(
4586:                 rule=rule, passed=True, observed_value=value,
4587:                 expected_value=None, violation_severity=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L115_56", 0.0), penalty_applied=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L115_77", 0.0)
4588:             )
4589:
4590:         min_val, max_val = rule.expected_range
4591:         passed = min_val <= value <= max_val
4592:
4593:         if not passed:
4594:             # Calculate violation severity based on how far outside range
4595:             if value < min_val:
4596:                 violation_severity = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L124_41", 1.0), (min_val - value) / max(abs(min_val), ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L124_84", 1.0)))
4597:             else:
4598:                 violation_severity = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L126_41", 1.0), (value - max_val) / max(abs(max_val), ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L126_84", 1.0)))
4599:             else:

```

```
4600:         violation_severity = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "violation_severity", 0.0) # Refactored
4601:
4602:         penalty = violation_severity * rule.penalty_factor if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_range", "auto_param_L130_78", 0.0)
4603:
4604:         return ValidationResult(
4605:             rule=rule, passed=passed, observed_value=value,
4606:             expected_value=rule.expected_range, violation_severity=violation_severity,
4607:             penalty_applied=penalty
4608:         )
4609:
4610: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit")
4611: def validate_unit(self, unit: str, rule: ValidationRule) -> ValidationResult:
4612:     """Validate unit matches expected unit"""
4613:     if rule.expected_unit is None:
4614:         return ValidationResult(
4615:             rule=rule, passed=True, observed_value=unit,
4616:             expected_value=None, violation_severity=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit", "auto_param_L144_56", 0.0), penalty_applied=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit", "auto_param_L144_77", 0.0)
4617:         )
4618:
4619:     passed = unit.lower() == rule.expected_unit.lower()
4620:     violation_severity = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit", "auto_param_L148_29", 1.0) if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit", "auto_param_L148_52", 0.0)
4621:     penalty = violation_severity * rule.penalty_factor if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_unit", "auto_param_L149_78", 0.0)
4622:
4623:     return ValidationResult(
4624:         rule=rule, passed=passed, observed_value=unit,
4625:         expected_value=rule.expected_unit, violation_severity=violation_severity,
4626:         penalty_applied=penalty
4627:     )
4628:
4629: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period")
4630: def validate_period(self, period: str, rule: ValidationRule) -> ValidationResult:
4631:     """Validate temporal period matches expected period"""
4632:     if rule.expected_period is None:
4633:         return ValidationResult(
4634:             rule=rule, passed=True, observed_value=period,
4635:             expected_value=None, violation_severity=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period", "auto_param_L163_56", 0.0), penalty_applied=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period", "auto_param_L163_77", 0.0)
4636:         )
4637:
4638:     passed = period.lower() == rule.expected_period.lower()
4639:     violation_severity = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period", "auto_param_L167_52", 1.0) if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period", "auto_param_L167_52", 0.0)
4640:     penalty = violation_severity * rule.penalty_factor if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_period", "auto_param_L168_78", 0.0)
4641:
4642:     return ValidationResult(
4643:         rule=rule, passed=passed, observed_value=period,
```

```
4644:         expected_value=rule.expected_period, violation_severity=violation_severity,
4645:         penalty_applied=penalty
4646:     )
4647:
4648:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity")
4649:     def validate_entity(self, entity: str, rule: ValidationRule) -> ValidationResult:
4650:         """Validate entity matches expected entity"""
4651:         if rule.expected_entity is None:
4652:             return ValidationResult(
4653:                 rule=rule, passed=True, observed_value=entity,
4654:                 expected_value=None, violation_severity=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity", "auto_param_L182_56", 0.0), penalty_applied=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity", "auto_param_L182_77", 0.0)
4655:             )
4656:
4657:         passed = entity.lower() == rule.expected_entity.lower()
4658:         violation_severity = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity", "auto_param_L186_29", 1.0) if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity", "auto_param_L186_52", 0.0)
4659:         penalty = violation_severity * rule.penalty_factor if not passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_entity", "auto_param_L187_78", 0.0)
4660:
4661:         return ValidationResult(
4662:             rule=rule, passed=passed, observed_value=entity,
4663:             expected_value=rule.expected_entity, violation_severity=violation_severity,
4664:             penalty_applied=penalty
4665:         )
4666:
4667:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.validate_data")
4668:     def validate_data(self, data: dict[str, Any]) -> list[ValidationResult]:
4669:         """Validate data against all rules"""
4670:         results = []
4671:
4672:         for rule in self.rules:
4673:             if rule.field_name not in data:
4674:                 continue
4675:
4676:             value = data[rule.field_name]
4677:
4678:             if rule.validator_type == ValidatorType.RANGE:
4679:                 result = self.validate_range(value, rule)
4680:             elif rule.validator_type == ValidatorType.UNIT:
4681:                 result = self.validate_unit(value, rule)
4682:             elif rule.validator_type == ValidatorType.PERIOD:
4683:                 result = self.validate_period(value, rule)
4684:             elif rule.validator_type == ValidatorType.ENTITY:
4685:                 result = self.validate_entity(value, rule)
4686:             else:
4687:                 continue
4688:
4689:             results.append(result)
4690:
4691:         return results
4692:
4693:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ReconciliationValidator.calculate_total_penalty")
4694:     def calculate_total_penalty(self, validation_results: list[ValidationResult]) -> float:
```

```
4695:         """Calculate total penalty from validation results"""
4696:         return sum(r.penalty_applied for r in validation_results)
4697:
4698: # =====
4699: # MICRO LEVEL: BAYESIAN UPDATER
4700: # =====
4701:
4702: @dataclass
4703: class ProbativeTest:
4704:     """Definition of a probative test"""
4705:     test_type: ProbativeTestType
4706:     test_name: str
4707:     evidence_strength: float # How strong the evidence if test passes
4708:     prior_probability: float # Prior belief before test
4709:
4710:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio")
4711:     def calculate_likelihood_ratio(self, test_passed: bool) -> float:
4712:         """
4713:         Calculate Bayesian likelihood ratio
4714:
4715:         Straw-in-wind: weak confirmation (LR ~ 2)
4716:         Hoop test: strong disconfirmation if fails (LR ~ ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_like
4717:         Smoking gun: strong confirmation if passes (LR ~ 10)
4718:         Doubly decisive: both necessary and sufficient (LR ~ 20 if passes, ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeT
4719:         est.calculate_likelihood_ratio", "auto_param_L246_75", 0.05) if fails)
4720:         """
4721:         if self.test_type == ProbativeTestType.STRAW_IN_WIND:
4722:             return 2.0 if test_passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio",
4723:             "auto_param_L249_43", 0.8)
4724:         elif self.test_type == ProbativeTestType.HOOP_TEST:
4725:             return 1.2 if test_passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio",
4726:             "auto_param_L251_43", 0.1)
4727:         elif self.test_type == ProbativeTestType.SMOKING_GUN:
4728:             return 1.0 if test_passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio",
4729:             "auto_param_L253_44", 0.9)
4730:         elif self.test_type == ProbativeTestType.DOUBLY_DECISIVE:
4731:             return 2.0 if test_passed else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio",
4732:             "auto_param_L255_44", 0.05)
4733:         else:
4734:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ProbativeTest.calculate_likelihood_ratio", "auto_param_L257_19", 1
4735:             .0)
4736:
4737: @dataclass
4738: class BayesianUpdate:
4739:     """Result of Bayesian updating"""
4740:     test: ProbativeTest
4741:     test_passed: bool
4742:     prior: float
4743:     likelihood_ratio: float
4744:     posterior: float
4745:     evidence_weight: float
4746:
4747: class BayesianUpdater:
4748:     """
4749:     Bayesian Updater: Sequential Bayesian updating based on probative test taxonomy
4750:     """
```

```
4744: Generates posterior_table_micro.csv
4745: """
4746:
4747: def __init__(self) -> None:
4748:     self.logger = logging.getLogger(self.__class__.__name__)
4749:     self.updates: list[BayesianUpdate] = []
4750:
4751: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.BayesianUpdater.update")
4752: def update(self, prior: float, test: ProbativeTest, test_passed: bool) -> float:
4753:     """
4754:     Perform Bayesian update using probative test
4755:
4756:      $P(H|E) = P(E|H) * P(H) / P(E)$ 
4757:
4758:     Using odds form:
4759:      $O(H|E) = LR * O(H)$ 
4760:     """
4761:     # Calculate likelihood ratio
4762:     lr = test.calculate_likelihood_ratio(test_passed)
4763:
4764:     # Convert prior probability to odds
4765:     prior_odds = prior / (1 - prior + 1e-10)
4766:
4767:     # Update odds
4768:     posterior_odds = lr * prior_odds
4769:
4770:     # Convert back to probability
4771:     posterior = posterior_odds / (1 + posterior_odds)
4772:
4773:     # Ensure valid probability
4774:     posterior = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianUpdater.update", "auto_param_L302_24", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianUpdater.update", "auto_param_L302_33", 1.0), posterior))
4775:
4776:     # Calculate evidence weight (KL divergence)
4777:     evidence_weight = self._calculate_evidence_weight(prior, posterior)
4778:
4779:     # Record update
4780:     update = BayesianUpdate(
4781:         test=test,
4782:         test_passed=test_passed,
4783:         prior=prior,
4784:         likelihood_ratio=lr,
4785:         posterior=posterior,
4786:         evidence_weight=evidence_weight
4787:     )
4788:     self.updates.append(update)
4789:
4790:     self.logger.debug(
4791:         f"Bayesian update: {test.test_name} ({test.test_type.value}): "
4792:         f"prior={prior:.3f} &206222 posterior={posterior:.3f} (LR={lr:.2f})"
4793:     )
4794:
4795:     return posterior
4796:
4797: def sequential_update(
4798:     self,
```

```
4799:         initial_prior: float,
4800:         tests: list[tuple[ProbativeTest, bool]]
4801:     ) -> float:
4802:         """Sequentially update belief through multiple tests"""
4803:         current_belief = initial_prior
4804:
4805:         for test, test_passed in tests:
4806:             current_belief = self.update(current_belief, test, test_passed)
4807:
4808:         return current_belief
4809:
4810: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.BayesianUpdater._calculate_evidence_weight")
4811: def _calculate_evidence_weight(self, prior: float, posterior: float) -> float:
4812:     """Calculate evidence weight using KL divergence"""
4813:     # Avoid log(0)
4814:     prior = max(1e-10, min(1 - 1e-10, prior))
4815:     posterior = max(1e-10, min(1 - 1e-10, posterior))
4816:
4817:     # KL divergence: D_KL(posterior || prior)
4818:     kl_div = (
4819:         posterior * np.log(posterior / prior) +
4820:         (1 - posterior) * np.log((1 - posterior) / (1 - prior))
4821:     )
4822:
4823:     return abs(kl_div)
4824:
4825: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.BayesianUpdater.export_to_csv")
4826: def export_to_csv(self, output_path: Path) -> None:
4827:     """Export posterior table to CSV"""
4828:     # Delegate to factory for I/O operation
4829:     from farfan_pipeline.analysis.factory import write_csv
4830:
4831:     headers = [
4832:         'test_name', 'test_type', 'test_passed', 'prior',
4833:         'likelihood_ratio', 'posterior', 'evidence_weight'
4834:     ]
4835:
4836:     rows = []
4837:     for update in self.updates:
4838:         rows.append([
4839:             update.test.test_name,
4840:             update.test.test_type.value,
4841:             update.test_passed,
4842:             f"{update.prior:.4f}",
4843:             f"{update.likelihood_ratio:.4f}",
4844:             f"{update.posterior:.4f}",
4845:             f"{update.evidence_weight:.4f}"
4846:         ])
4847:
4848:     write_csv(rows, output_path, headers=headers)
4849:     self.logger.info(f"Exported {len(self.updates)} Bayesian updates to {output_path}")
4850:
4851: # =====
4852: # MICRO LEVEL: INTEGRATION
4853: # =====
4854:
```

```

4855: @dataclass
4856: class MicroLevelAnalysis:
4857:     """Complete micro-level analysis with reconciliation and Bayesian updating"""
4858:     question_id: str
4859:     raw_score: float
4860:     validation_results: list[ValidationResult]
4861:     validation_penalty: float
4862:     bayesian_updates: list[BayesianUpdate]
4863:     final_posterior: float
4864:     adjusted_score: float
4865:     metadata: dict[str, Any] = field(default_factory=dict)
4866:
4867: # =====
4868: # MESO LEVEL: DISPERSION ENGINE
4869: # =====
4870:
4871: class DispersionEngine:
4872:     """
4873:     Dispersion Engine: Computes CV, max_gap, Gini coefficient
4874:     Integrates dispersion penalties into meso-level scoring
4875:     """
4876:
4877:     def __init__(self, dispersion_threshold: float = 0.3) -> None:
4878:         self.dispersion_threshold = dispersion_threshold
4879:         self.logger = logging.getLogger(self.__class__.__name__)
4880:
4881:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_cv")
4882:     def calculate_cv(self, scores: list[float]) -> float:
4883:         """Calculate Coefficient of Variation (CV = std / mean)"""
4884:         if not scores or len(scores) < 2:
4885:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_cv", "auto_param_L413_19", 0.0)
4886:
4887:         mean_score = np.mean(scores)
4888:         std_score = np.std(scores, ddof=1)
4889:
4890:         if mean_score == 0:
4891:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_cv", "auto_param_L419_19", 0.0)
4892:
4893:         cv = std_score / mean_score
4894:         return cv
4895:
4896:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_max_gap")
4897:     def calculate_max_gap(self, scores: list[float]) -> float:
4898:         """Calculate maximum gap between adjacent scores"""
4899:         if not scores or len(scores) < 2:
4900:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_max_gap", "auto_param_L428_19", 0.0)
4901:
4902:         sorted_scores = sorted(scores)
4903:         gaps = [sorted_scores[i+1] - sorted_scores[i] for i in range(len(sorted_scores) - 1)]
4904:
4905:         return max(gaps) if gaps else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_max_gap", "auto_param_L433_38", 0.0)
4906:
4907:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_gini")
4908:     def calculate_gini(self, scores: list[float]) -> float:
4909:         """

```

```
4910:         Calculate Gini coefficient
4911:         0 = perfect equality, 1 = perfect inequality
4912:         """
4913:         if not scores or len(scores) < 2:
4914:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_gini", "auto_param_L442_19", 0.0)
4915:
4916:         # Sort scores
4917:         sorted_scores = np.array(sorted(scores))
4918:         n = len(sorted_scores)
4919:
4920:         # Calculate Gini
4921:         index = np.arange(1, n + 1)
4922:         gini = (2 * np.sum(index * sorted_scores)) / (n * np.sum(sorted_scores)) - (n + 1) / n
4923:
4924:         return gini
4925:
4926: @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty")
4927: def calculate_dispersion_penalty(self, scores: list[float]) -> tuple[float, dict[str, float]]:
4928:     """
4929:     Calculate dispersion penalty based on CV, max_gap, and Gini
4930:     Returns (penalty, metrics_dict)
4931:     """
4932:     cv = self.calculate_cv(scores)
4933:     max_gap = self.calculate_max_gap(scores)
4934:     gini = self.calculate_gini(scores)
4935:
4936:     # Calculate penalties for each metric
4937:     cv_penalty = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L465_25", 0.0), (cv - self.dispersion_threshold) * ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L465_65", 0.5))
4938:     gap_penalty = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L466_26", 0.0), (max_gap - ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L466_42", 1.0)) * ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L466_49", 0.3)) #
4939:     Penalty if gap > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L466_74", 1.0)
4939:     gini_penalty = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L467_27", 0.0), (gini - ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L467_40", 0.3)) * ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L467_47", 0.4)) # P
4940:     enalty if Gini > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L467_73", 0.3)
4940:
4941:     # Total penalty (capped at ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L469_35", 1.0))
4942:     total_penalty = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.DispersionEngine.calculate_dispersion_penalty", "auto_param_L470_28", 1.0), cv_penalty + gap_penalty + gini_penalty)
4943:
4944:     metrics = {
4945:         'cv': cv,
4946:         'max_gap': max_gap,
4947:         'gini': gini,
4948:         'cv_penalty': cv_penalty,
4949:         'gap_penalty': gap_penalty,
4950:         'gini_penalty': gini_penalty,
4951:         'total_penalty': total_penalty
4952:     }
4953:
4954:     self.logger.debug(
4955:         f"Dispersion metrics: CV={cv:.3f}, max_gap={max_gap:.3f}, "
```

```
4956:         f"Gini={gini:.3f}, penalty={total_penalty:.3f}"
4957:     )
4958:
4959:     return total_penalty, metrics
4960:
4961: # =====
4962: # MESO LEVEL: PEER CALIBRATION
4963: # =====
4964:
4965: @dataclass
4966: class PeerContext:
4967:     """Peer context for comparison"""
4968:     peer_id: str
4969:     peer_name: str
4970:     scores: dict[str, float] # dimension -> score
4971:     metadata: dict[str, Any] = field(default_factory=dict)
4972:
4973: @dataclass
4974: class PeerComparison:
4975:     """Result of peer comparison"""
4976:     target_score: float
4977:     peer_mean: float
4978:     peer_std: float
4979:     z_score: float
4980:     percentile: float
4981:     deviation_penalty: float
4982:     narrative: str
4983:
4984: class PeerCalibrator:
4985:     """
4986:     Peer Calibration: Compare scores against peer context
4987:     Generate narrative hooks for contextualization
4988:     """
4989:
4990:     def __init__(self, deviation_threshold: float = 1.5) -> None:
4991:         self.deviation_threshold = deviation_threshold # Z-score threshold
4992:         self.logger = logging.getLogger(self.__class__.__name__)
4993:
4994:     def compare_to_peers(
4995:         self,
4996:         target_score: float,
4997:         peer_contexts: list[PeerContext],
4998:         dimension: str
4999:     ) -> PeerComparison:
5000:         """Compare target score to peer contexts"""
5001:         # Extract peer scores for this dimension
5002:         peer_scores = [
5003:             peer.scores.get(dimension, ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L531_39"
5004: , 0.0))
5005:             for peer in peer_contexts
5006:             if dimension in peer.scores
5007:         ]
5008:
5009:         if not peer_scores:
5010:             return PeerComparison(
5011:                 target_score=target_score,
```

```
5011:         peer_mean=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L539_26", 0.0),
5012:         peer_std=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L540_25", 0.0),
5013:         z_score=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L541_24", 0.0),
5014:         percentile=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L542_27", 0.5),
5015:         deviation_penalty=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L543_34", 0.0
),
5016:         narrative="No peer data available for comparison"
5017:     )
5018:
5019:     # Calculate peer statistics
5020:     peer_mean = np.mean(peer_scores)
5021:     peer_std = np.std(peer_scores, ddof=1) if len(peer_scores) > 1 else ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalib
rator.__init__", "auto_param_L549_76", 1.0)
5022:
5023:     # Calculate z-score
5024:     z_score = (target_score - peer_mean) / (peer_std + 1e-10)
5025:
5026:     # Calculate percentile
5027:     percentile = stats.percentileofscore(peer_scores, target_score) / 100.0
5028:
5029:     # Calculate deviation penalty
5030:     deviation_penalty = max(0.0, (abs(z_score) - self.deviation_threshold) * 0.2)
5031:     deviation_penalty = min(0.5, deviation_penalty)
5032:
5033:     # Generate narrative
5034:     narrative = self._generate_narrative(
5035:         target_score, peer_mean, peer_std, z_score, percentile
5036:     )
5037:
5038:     return PeerComparison(
5039:         target_score=target_score,
5040:         peer_mean=peer_mean,
5041:         peer_std=peer_std,
5042:         z_score=z_score,
5043:         percentile=percentile,
5044:         deviation_penalty=deviation_penalty,
5045:         narrative=narrative
5046:     )
5047:
5048:     def _generate_narrative(
5049:         self,
5050:         score: float,
5051:         peer_mean: float,
5052:         peer_std: float,
5053:         z_score: float,
5054:         percentile: float
5055:     ) -> str:
5056:         """Generate narrative hook for peer comparison"""
5057:         # Determine performance relative to peers
5058:         if z_score > 1.5:
5059:             performance = "significantly above"
5060:         elif z_score > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L588_23", 0.5):
5061:             performance = "moderately above"
5062:         elif z_score > -ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L590_24", 0.5):
5063:             performance = "comparable to"
5064:         elif z_score > -1.5:
```

```
5065:         performance = "moderately below"
5066:     else:
5067:         performance = "significantly below"
5068:
5069:     # Determine percentile description
5070:     if percentile >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L598_25", 0.9):
5071:         rank = "top 10%"
5072:     elif percentile >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L600_27", 0.75):
5073:         rank = "top quartile"
5074:     elif percentile >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L602_27", 0.5):
5075:         rank = "above median"
5076:     elif percentile >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.PeerCalibrator.__init__", "auto_param_L604_27", 0.25):
5077:         rank = "below median"
5078:     else:
5079:         rank = "bottom quartile"
5080:
5081:     narrative = (
5082:         f"Score of {score:.2f} is {performance} peer average "
5083:         f"({peer_mean:.2f} ± {peer_std:.2f}), "
5084:         f"placing in the {rank} (percentile: {percentile:.1%})"
5085:     )
5086:
5087:     return narrative
5088:
5089: # =====
5090: # MESO LEVEL: BAYESIAN ROLL-UP
5091: # =====
5092:
5093: @dataclass
5094: class MesoLevelAnalysis:
5095:     """Complete meso-level analysis with dispersion and peer calibration"""
5096:     cluster_id: str
5097:     micro_scores: list[float]
5098:     raw_meso_score: float
5099:     dispersion_metrics: dict[str, float]
5100:     dispersion_penalty: float
5101:     peer_comparison: PeerComparison | None
5102:     peer_penalty: float
5103:     total_penalty: float
5104:     final_posterior: float
5105:     adjusted_score: float
5106:     metadata: dict[str, Any] = field(default_factory=dict)
5107:
5108: class BayesianRollUp:
5109:     """
5110:     Bayesian Roll-Up: Aggregate micro posteriors to meso level with penalties
5111:     """
5112:
5113:     def __init__(self) -> None:
5114:         self.logger = logging.getLogger(self.__class__.__name__)
5115:
5116:     def aggregate_micro_to_meso(
5117:         self,
5118:         micro_analyses: list[MicroLevelAnalysis],
5119:         dispersion_penalty: float = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__", "auto_param_L647_36", 0
5120:     ),
```

```
5120:         peer_penalty: float = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__", "auto_param_L648_30", 0.0),
5121:         additional_penalties: dict[str, float] | None = None
5122:     ) -> float:
5123:         """
5124:         Aggregate micro-level posteriors to meso-level posterior
5125:
5126:         Uses hierarchical Bayesian model:
5127:         - Micro posteriors are observations
5128:         - Meso posterior is hyperparameter
5129:         """
5130:         if not micro_analyses:
5131:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__", "auto_param_L659_19", 0.0)
5132:
5133:         # Extract posteriors (use micro-level adjusted scores so reconciliation
5134:         # penalties propagate into the meso aggregation)
5135:         posteriors = [m.adjusted_score for m in micro_analyses]
5136:
5137:         # Calculate weighted mean (could use Beta-Binomial hierarchical model)
5138:         raw_meso_posterior = np.mean(posteriors)
5139:
5140:         # Apply penalties
5141:         total_penalty = dispersion_penalty + peer_penalty
5142:         if additional_penalties:
5143:             total_penalty += sum(additional_penalties.values())
5144:
5145:         # Adjust posterior (multiplicative penalty)
5146:         adjusted_posterior = raw_meso_posterior * (1 - total_penalty)
5147:         adjusted_posterior = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__", "auto_param_L675_33", 0.0),
, min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__", "auto_param_L675_42", 1.0), adjusted_posterior))
5148:
5149:         self.logger.debug(
5150:             f"Meso roll-up: {len(micro_analyses)} micro â\206\222 "
5151:             f"raw={raw_meso_posterior:.3f}, penalty={total_penalty:.3f}, "
5152:             f"adjusted={adjusted_posterior:.3f}"
5153:         )
5154:
5155:         return adjusted_posterior
5156:
5157:     def export_to_csv(
5158:         self,
5159:         meso_analyses: list[MesoLevelAnalysis],
5160:         output_path: Path
5161:     ) -> None:
5162:         """Export meso posterior table to CSV"""
5163:         # Delegate to factory for I/O operation
5164:         from farfan_pipeline.analysis.factory import write_csv
5165:
5166:         headers = [
5167:             'cluster_id', 'raw_meso_score', 'dispersion_penalty',
5168:             'peer_penalty', 'total_penalty', 'adjusted_score',
5169:             'cv', 'max_gap', 'gini'
5170:         ]
5171:
5172:         rows = []
5173:         for analysis in meso_analyses:
5174:             rows.append([
```

```
5175:         analysis.cluster_id,
5176:         f"{analysis.raw_meso_score:.4f}",
5177:         f"{analysis.dispersion_penalty:.4f}",
5178:         f"{analysis.peer_penalty:.4f}",
5179:         f"{analysis.total_penalty:.4f}",
5180:         f"{analysis.adjusted_score:.4f}",
5181:         f"{analysis.dispersion_metrics.get('cv', ParameterLoaderV2.get('farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__', "a
uto_param_L709_57", 0.0)):.4f}",
5182:         f"{analysis.dispersion_metrics.get('max_gap', ParameterLoaderV2.get('farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__
", "auto_param_L710_62", 0.0)):.4f}",
5183:         f"{analysis.dispersion_metrics.get('gini', ParameterLoaderV2.get('farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__',
"auto_param_L711_59", 0.0)):.4f}"
5184:     ])
5185:
5186:     write_csv(rows, output_path, headers=headers)
5187:
5188:     self.logger.info(
5189:         f"Exported {len(meso_analyses)} meso analyses to {output_path}"
5190:     )
5191:
5192: # =====
5193: # MACRO LEVEL: CONTRADICTION SCANNER
5194: # =====
5195:
5196: @dataclass
5197: class ContradictionDetection:
5198:     """Detected contradiction between levels"""
5199:     level_a: str # e.g., "micro:P1-D1-Q1"
5200:     level_b: str # e.g., "meso:CL01"
5201:     score_a: float
5202:     score_b: float
5203:     discrepancy: float
5204:     severity: float # ParameterLoaderV2.get('farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__', "auto_param_L732_23", 0.0)-Parameter
LoaderV2.get('farfan_core.analysis.bayesian_multilevel_system.BayesianRollUp.__init__', "auto_param_L732_27", 1.0)
5205:     description: str
5206:
5207: class ContradictionScanner:
5208:     """
5209:     Macro Contradiction Scanner: Detect inconsistencies between micro and meso levels
5210:     """
5211:
5212:     def __init__(self, discrepancy_threshold: float = 0.3) -> None:
5213:         self.discrepancy_threshold = discrepancy_threshold
5214:         self.logger = logging.getLogger(self.__class__.__name__)
5215:         self.contradictions: list[ContradictionDetection] = []
5216:
5217:     def scan_micro_meso(
5218:         self,
5219:         micro_analyses: list[MicroLevelAnalysis],
5220:         meso_analysis: MesoLevelAnalysis
5221:     ) -> list[ContradictionDetection]:
5222:         """Scan for contradictions between micro and meso levels"""
5223:         contradictions = []
5224:
5225:         for micro in micro_analyses:
5226:             discrepancy = abs(micro.adjusted_score - meso_analysis.adjusted_score)
```

```
5227:
5228:         if discrepancy > self.discrepancy_threshold:
5229:             severity = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ContradictionScanner.__init__", "auto_param_L757_31",
1.0), discrepancy / 2.0)
5230:
5231:             contradiction = ContradictionDetection(
5232:                 level_a=f"micro:{micro.question_id}",
5233:                 level_b=f"meso:{meso_analysis.cluster_id}",
5234:                 score_a=micro.adjusted_score,
5235:                 score_b=meso_analysis.adjusted_score,
5236:                 discrepancy=discrepancy,
5237:                 severity=severity,
5238:                 description=f"Micro question {micro.question_id} score "
5239:                     f"({micro.adjusted_score:.2f}) differs significantly from "
5240:                     f"meso cluster {meso_analysis.cluster_id} "
5241:                     f"({meso_analysis.adjusted_score:.2f})"
5242:             )
5243:
5244:             contradictions.append(contradiction)
5245:             self.contradictions.append(contradiction)
5246:
5247:         return contradictions
5248:
5249:     def scan_meso_macro(
5250:         self,
5251:         meso_analyses: list[MesoLevelAnalysis],
5252:         macro_score: float
5253:     ) -> list[ContradictionDetection]:
5254:         """Scan for contradictions between meso and macro levels"""
5255:         contradictions = []
5256:
5257:         for meso in meso_analyses:
5258:             discrepancy = abs(meso.adjusted_score - macro_score)
5259:
5260:             if discrepancy > self.discrepancy_threshold:
5261:                 severity = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ContradictionScanner.__init__", "auto_param_L789_31",
1.0), discrepancy / 2.0)
5262:
5263:                 contradiction = ContradictionDetection(
5264:                     level_a=f"meso:{meso.cluster_id}",
5265:                     level_b="macro:overall",
5266:                     score_a=meso.adjusted_score,
5267:                     score_b=macro_score,
5268:                     discrepancy=discrepancy,
5269:                     severity=severity,
5270:                     description=f"Meso cluster {meso.cluster_id} score "
5271:                         f"({meso.adjusted_score:.2f}) differs significantly from "
5272:                         f"macro overall ({macro_score:.2f})"
5273:                 )
5274:
5275:                 contradictions.append(contradiction)
5276:                 self.contradictions.append(contradiction)
5277:
5278:         return contradictions
5279:
5280:     @calibrated_method("farfan_core.analysis.bayesian_multilevel_system.ContradictionScanner.calculate_contradiction_penalty")
```

```

5281:     def calculate_contradiction_penalty(self) -> float:
5282:         """Calculate penalty based on detected contradictions"""
5283:         if not self.contradictions:
5284:             return ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.ContradictionScanner.calculate_contradiction_penalty", "auto_param
_L812_19", 0.0)
5285:
5286:         # Average severity weighted by number of contradictions
5287:         avg_severity = np.mean([c.severity for c in self.contradictions])
5288:         count_factor = min(1.0, len(self.contradictions) / 10.0)
5289:
5290:         penalty = avg_severity * count_factor * 0.5
5291:
5292:         return penalty
5293:
5294: # =====
5295: # MACRO LEVEL: BAYESIAN PORTFOLIO COMPOSER
5296: # =====
5297:
5298: @dataclass
5299: class MacroLevelAnalysis:
5300:     """Complete macro-level portfolio analysis"""
5301:     overall_posterior: float
5302:     coverage_score: float
5303:     coverage_penalty: float
5304:     dispersion_score: float
5305:     dispersion_penalty: float
5306:     contradiction_count: int
5307:     contradiction_penalty: float
5308:     total_penalty: float
5309:     adjusted_score: float
5310:     cluster_scores: dict[str, float]
5311:     recommendations: list[str]
5312:     metadata: dict[str, Any] = field(default_factory=dict)
5313:
5314: class BayesianPortfolioComposer:
5315:     """
5316:     Macro Bayesian Portfolio Composer:
5317:     Aggregate all evidence with coverage, dispersion, and contradiction penalties
5318:     """
5319:
5320:     def __init__(self) -> None:
5321:         self.logger = logging.getLogger(self.__class__.__name__)
5322:
5323:     def calculate_coverage(
5324:         self,
5325:         questions_answered: int,
5326:         total_questions: int
5327:     ) -> tuple[float, float]:
5328:         """
5329:         Calculate coverage score and penalty
5330:         Returns (coverage_score, penalty)
5331:         """
5332:         coverage = questions_answered / max(total_questions, 1)
5333:
5334:         # Penalty increases sharply below 70% coverage
5335:         if coverage >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L863_23", 0.9

```

```
):
5336:         penalty = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "penalty", 0.0) # Refactor
ed
5337:         elif coverage >= ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L865_25", 0
.7):
5338:             penalty = (ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L866_23", 0.9
) - coverage) * ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L866_41", 0.5)
5339:         else:
5340:             penalty = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L868_22", 0.1)
+ (ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L868_29", 0.7) - coverage) * ParameterL
oaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L868_47", 1.0)
5341:
5342:         penalty = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L870_22", 1.0)
, penalty)
5343:
5344:         return coverage, penalty
5345:
5346:     def compose_macro_portfolio(
5347:         self,
5348:         meso_analyses: list[MesoLevelAnalysis],
5349:         total_questions: int,
5350:         contradiction_scanner: ContradictionScanner
5351:     ) -> MacroLevelAnalysis:
5352:         """
5353:         Compose macro-level portfolio from meso analyses
5354:         """
5355:         if not meso_analyses:
5356:             return MacroLevelAnalysis(
5357:                 overall_posterior=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L8
85_34", 0.0),
5358:                 coverage_score=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L886_
31", 0.0),
5359:                 coverage_penalty=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L88
7_33", 1.0),
5360:                 dispersion_score=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L88
8_33", 0.0),
5361:                 dispersion_penalty=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L
889_35", 0.0),
5362:                 contradiction_count=0,
5363:                 contradiction_penalty=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_para
m_L891_38", 0.0),
5364:                 total_penalty=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L892_3
0", 1.0),
5365:                 adjusted_score=ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L893_
31", 0.0),
5366:                 cluster_scores={},
5367:                 recommendations=["No meso-level data available"]
5368:             )
5369:
5370:         # Calculate raw overall posterior (mean of meso scores)
5371:         meso_scores = [m.adjusted_score for m in meso_analyses]
5372:         raw_overall = np.mean(meso_scores)
5373:
5374:         # Calculate coverage
5375:         questions_answered = sum(len(m.micro_scores) for m in meso_analyses)
5376:         coverage_score, coverage_penalty = self.calculate_coverage(
```

```
5377:         questions_answered, total_questions
5378:     )
5379:
5380:     # Calculate portfolio-level dispersion
5381:     dispersion_engine = DispersionEngine()
5382:     dispersion_penalty, dispersion_metrics = dispersion_engine.calculate_dispersion_penalty(meso_scores)
5383:     dispersion_score = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L911_27",
1.0) - dispersion_penalty
5384:
5385:     # Get contradiction penalty
5386:     contradiction_penalty = contradiction_scanner.calculate_contradiction_penalty()
5387:     contradiction_count = len(contradiction_scanner.contradictions)
5388:
5389:     # Total penalty
5390:     total_penalty = coverage_penalty + dispersion_penalty + contradiction_penalty
5391:     total_penalty = min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L919_28",
, 1.0), total_penalty)
5392:
5393:     # Adjusted score
5394:     adjusted_score = raw_overall * (1 - total_penalty)
5395:     adjusted_score = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L923_29", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L923_38", 1.0), adjusted_score
))
5396:
5397:     # Extract cluster scores
5398:     cluster_scores = {m.cluster_id: m.adjusted_score for m in meso_analyses}
5399:
5400:     # Generate recommendations
5401:     recommendations = self._generate_recommendations(
5402:         coverage_score, dispersion_score, contradiction_count,
5403:         coverage_penalty, dispersion_penalty, contradiction_penalty
5404:     )
5405:
5406:     self.logger.info(
5407:         f"Macro portfolio: raw={raw_overall:.3f}, "
5408:         f"coverage_pen={coverage_penalty:.3f}, "
5409:         f"dispersion_pen={dispersion_penalty:.3f}, "
5410:         f"contradiction_pen={contradiction_penalty:.3f}, "
5411:         f"final={adjusted_score:.3f}"
5412:     )
5413:
5414:     return MacroLevelAnalysis(
5415:         overall_posterior=raw_overall,
5416:         coverage_score=coverage_score,
5417:         coverage_penalty=coverage_penalty,
5418:         dispersion_score=dispersion_score,
5419:         dispersion_penalty=dispersion_penalty,
5420:         contradiction_count=contradiction_count,
5421:         contradiction_penalty=contradiction_penalty,
5422:         total_penalty=total_penalty,
5423:         adjusted_score=adjusted_score,
5424:         cluster_scores=cluster_scores,
5425:         recommendations=recommendations,
5426:         metadata={
5427:             'dispersion_metrics': dispersion_metrics,
5428:             'questions_answered': questions_answered,
```

```
5429:         'total_questions': total_questions
5430:     }
5431: )
5432:
5433: def _generate_recommendations(
5434:     self,
5435:     coverage: float,
5436:     dispersion: float,
5437:     contradiction_count: int,
5438:     coverage_penalty: float,
5439:     dispersion_penalty: float,
5440:     contradiction_penalty: float
5441: ) -> list[str]:
5442:     """Generate strategic recommendations based on portfolio analysis"""
5443:     recommendations = []
5444:
5445:     if coverage_penalty > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L973_3
0", 0.1):
5446:         recommendations.append(
5447:             f"Improve question coverage (current: {coverage:.1%}). "
5448:             "Address unanswered questions to reduce coverage penalty."
5449:         )
5450:
5451:     if dispersion_penalty > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L979
_32", 0.1):
5452:         recommendations.append(
5453:             f"Reduce score dispersion across clusters (current penalty: {dispersion_penalty:.2f}). "
5454:             "Focus on bringing lower-performing areas up to standard."
5455:         )
5456:
5457:     if contradiction_penalty > ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L
985_35", 0.05):
5458:         recommendations.append(
5459:             f"Resolve {contradiction_count} detected contradictions between levels. "
5460:             "Ensure consistency in assessment across micro/meso/macro."
5461:         )
5462:
5463:     if not recommendations:
5464:         recommendations.append(
5465:             "Portfolio is well-balanced with good coverage, low dispersion, "
5466:             "and minimal contradictions. Continue current approach."
5467:         )
5468:
5469:     return recommendations
5470:
5471: def export_to_csv(
5472:     self,
5473:     macro_analysis: MacroLevelAnalysis,
5474:     output_path: Path
5475: ) -> None:
5476:     """Export macro posterior table to CSV"""
5477:     # Delegate to factory for I/O operation
5478:     from farfan_pipeline.analysis.factory import write_csv
5479:
5480:     headers = ['metric', 'value', 'penalty', 'description']
5481:
```

```
5482:         rows = [
5483:             [
5484:                 'overall_posterior',
5485:                 f"{macro_analysis.overall_posterior:.4f}",
5486:                 f"{macro_analysis.total_penalty:.4f}",
5487:                 'Raw overall score before penalties'
5488:             ],
5489:             [
5490:                 'coverage',
5491:                 f"{macro_analysis.coverage_score:.4f}",
5492:                 f"{macro_analysis.coverage_penalty:.4f}",
5493:                 'Question coverage ratio'
5494:             ],
5495:             [
5496:                 'dispersion',
5497:                 f"{macro_analysis.dispersion_score:.4f}",
5498:                 f"{macro_analysis.dispersion_penalty:.4f}",
5499:                 'Portfolio dispersion score'
5500:             ],
5501:             [
5502:                 'contradictions',
5503:                 str(macro_analysis.contradiction_count),
5504:                 f"{macro_analysis.contradiction_penalty:.4f}",
5505:                 'Number of detected contradictions'
5506:             ],
5507:             [
5508:                 'adjusted_score',
5509:                 f"{macro_analysis.adjusted_score:.4f}",
5510:                 'ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L1038_17", 0.0000)'
5511:             ],
5512:             'Final penalty-adjusted score'
5513:         ]
5514:
5515:         write_csv(rows, output_path, headers=headers)
5516:         self.logger.info(f"Exported macro analysis to {output_path}")
5517:
5518: # =====
5519: # ORCHESTRATOR: COMPLETE MULTI-LEVEL PIPELINE
5520: # =====
5521:
5522: class MultiLevelBayesianOrchestrator:
5523:     """
5524:     Complete orchestration of microâ\206\222mesoâ\206\222macro Bayesian analysis pipeline
5525:     """
5526:
5527:     def __init__(
5528:         self,
5529:         validation_rules: list[ValidationRule],
5530:         output_dir: Path = Path("data/bayesian_outputs")
5531:     ) -> None:
5532:         self.validation_rules = validation_rules
5533:         self.output_dir = output_dir
5534:         self.output_dir.mkdir(parents=True, exist_ok=True)
5535:
5536:         # Initialize components
```

```
5537:         self.reconciliation_validator = ReconciliationValidator(validation_rules)
5538:         self.bayesian_updater = BayesianUpdater()
5539:         self.dispersion_engine = DispersionEngine()
5540:         self.peer_calibrator = PeerCalibrator()
5541:         self.bayesian_rollup = BayesianRollUp()
5542:         self.contradiction_scanner = ContradictionScanner()
5543:         self.portfolio_composer = BayesianPortfolioComposer()
5544:
5545:         self.logger = logging.getLogger(self.__class__.__name__)
5546:
5547:     def run_complete_analysis(
5548:         self,
5549:         micro_data: list[dict[str, Any]],
5550:         cluster_mapping: dict[str, list[str]], # cluster_id -> question_ids
5551:         peer_contexts: list[PeerContext] | None = None,
5552:         total_questions: int = 300
5553:     ) -> tuple[list[MicroLevelAnalysis], list[MesoLevelAnalysis], MacroLevelAnalysis]:
5554:         """
5555:         Run complete multi-level Bayesian analysis
5556:
5557:         Returns: (micro_analyses, meso_analyses, macro_analysis)
5558:         """
5559:         self.logger.info("=" * 80)
5560:         self.logger.info("MULTI-LEVEL BAYESIAN ANALYSIS PIPELINE")
5561:         self.logger.info("=" * 80)
5562:
5563:         # MICRO LEVEL
5564:         self.logger.info("\n[1/3] MICRO LEVEL: Reconciliation + Bayesian Updating")
5565:         micro_analyses = self._run_micro_level(micro_data)
5566:
5567:         # Export micro posteriors
5568:         self.bayesian_updater.export_to_csv(
5569:             self.output_dir / "posterior_table_micro.csv"
5570:         )
5571:
5572:         # MESO LEVEL
5573:         self.logger.info("\n[2/3] MESO LEVEL: Dispersion + Peer Calibration + Roll-Up")
5574:         meso_analyses = self._run_meso_level(
5575:             micro_analyses, cluster_mapping, peer_contexts
5576:         )
5577:
5578:         # Export meso posteriors
5579:         self.bayesian_rollup.export_to_csv(
5580:             meso_analyses,
5581:             self.output_dir / "posterior_table_meso.csv"
5582:         )
5583:
5584:         # MACRO LEVEL
5585:         self.logger.info("\n[3/3] MACRO LEVEL: Contradiction Scan + Portfolio Composition")
5586:         macro_analysis = self._run_macro_level(
5587:             micro_analyses, meso_analyses, total_questions
5588:         )
5589:
5590:         # Export macro posteriors
5591:         self.portfolio_composer.export_to_csv(
5592:             macro_analysis,
```

```
5593:         self.output_dir / "posterior_table_macro.csv"
5594:     )
5595:
5596:     self.logger.info("\n" + "=" * 80)
5597:     self.logger.info("ANALYSIS COMPLETE")
5598:     self.logger.info(f"Final adjusted score: {macro_analysis.adjusted_score:.4f}")
5599:     self.logger.info(f"Outputs saved to: {self.output_dir}")
5600:     self.logger.info("=" * 80)
5601:
5602:     return micro_analyses, meso_analyses, macro_analysis
5603:
5604:     def _run_micro_level(
5605:         self,
5606:         micro_data: list[dict[str, Any]]
5607:     ) -> list[MicroLevelAnalysis]:
5608:         """Run micro-level analysis"""
5609:         micro_analyses = []
5610:
5611:         for data in micro_data:
5612:             question_id = data.get('question_id', 'UNKNOWN')
5613:             raw_score = data.get('raw_score', ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L1141_46", 0.0))
5614:
5615:             # Reconciliation validation
5616:             validation_results = self.reconciliation_validator.validate_data(data)
5617:             validation_penalty = self.reconciliation_validator.calculate_total_penalty(
5618:                 validation_results
5619:             )
5620:
5621:             # Bayesian updating (using probative tests)
5622:             tests = data.get('probative_tests', [])
5623:             if tests:
5624:                 initial_prior = raw_score
5625:                 final_posterior = self.bayesian_updater.sequential_update(
5626:                     initial_prior, tests
5627:                 )
5628:             else:
5629:                 final_posterior = raw_score
5630:
5631:             # Calculate adjusted score
5632:             adjusted_score = final_posterior * (1 - validation_penalty)
5633:             adjusted_score = max(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L1161_33", 0.0), min(ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "auto_param_L1161_42", 1.0), adjusted_score))
5634:
5635:             analysis = MicroLevelAnalysis(
5636:                 question_id=question_id,
5637:                 raw_score=raw_score,
5638:                 validation_results=validation_results,
5639:                 validation_penalty=validation_penalty,
5640:                 bayesian_updates=self.bayesian_updater.updates[-len(tests):] if tests else [],
5641:                 final_posterior=final_posterior,
5642:                 adjusted_score=adjusted_score
5643:             )
5644:
5645:             micro_analyses.append(analysis)
```

```
5646:
5647:     self.logger.info(f"  Processed {len(micro_analyses)} micro-level questions")
5648:     return micro_analyses
5649:
5650:     def _run_meso_level(
5651:         self,
5652:         micro_analyses: list[MicroLevelAnalysis],
5653:         cluster_mapping: dict[str, list[str]],
5654:         peer_contexts: list[PeerContext] | None
5655:     ) -> list[MesoLevelAnalysis]:
5656:         """Run meso-level analysis"""
5657:         meso_analyses = []
5658:
5659:         for cluster_id, question_ids in cluster_mapping.items():
5660:             # Get micro analyses for this cluster
5661:             cluster_micros = [
5662:                 m for m in micro_analyses
5663:                 if m.question_id in question_ids
5664:             ]
5665:
5666:             if not cluster_micros:
5667:                 continue
5668:
5669:             # Get micro scores
5670:             micro_scores = [m.adjusted_score for m in cluster_micros]
5671:
5672:             # Calculate dispersion
5673:             dispersion_penalty, dispersion_metrics = (
5674:                 self.dispersion_engine.calculate_dispersion_penalty(micro_scores)
5675:             )
5676:
5677:             # Peer calibration
5678:             raw_meso_score = np.mean(micro_scores)
5679:             peer_comparison = None
5680:             peer_penalty = ParameterLoaderV2.get("farfan_core.analysis.bayesian_multilevel_system.BayesianPortfolioComposer.__init__", "peer_penalty", 0.0)
5681:
5682:             if peer_contexts:
5683:                 peer_comparison = self.peer_calibrator.compare_to_peers(
5684:                     raw_meso_score, peer_contexts, cluster_id
5685:                 )
5686:                 peer_penalty = peer_comparison.deviation_penalty
5687:
5688:             # Bayesian roll-up
5689:             adjusted_score = self.bayesian_rollup.aggregate_micro_to_meso(
5690:                 cluster_micros,
5691:                 dispersion_penalty,
5692:                 peer_penalty
5693:             )
5694:
5695:             total_penalty = dispersion_penalty + peer_penalty
5696:
5697:             analysis = MesoLevelAnalysis(
5698:                 cluster_id=cluster_id,
5699:                 micro_scores=micro_scores,
5700:                 raw_meso_score=raw_meso_score,
```

```
5701:         dispersion_metrics=dispersion_metrics,
5702:         dispersion_penalty=dispersion_penalty,
5703:         peer_comparison=peer_comparison,
5704:         peer_penalty=peer_penalty,
5705:         total_penalty=total_penalty,
5706:         final_posterior=adjusted_score,
5707:         adjusted_score=adjusted_score,
5708:         metadata={'question_ids': question_ids} # Add question_ids to metadata
5709:     )
5710:
5711:     meso_analyses.append(analysis)
5712:
5713:     self.logger.info(f"  Processed {len(meso_analyses)} meso-level clusters")
5714:     return meso_analyses
5715:
5716: def _run_macro_level(
5717:     self,
5718:     micro_analyses: list[MicroLevelAnalysis],
5719:     meso_analyses: list[MesoLevelAnalysis],
5720:     total_questions: int
5721: ) -> MacroLevelAnalysis:
5722:     """Run macro-level analysis"""
5723:     # Scan for contradictions
5724:     for meso in meso_analyses:
5725:         # Get question_ids for this meso cluster from metadata or empty list
5726:         meso_question_ids = meso.metadata.get('question_ids', [])
5727:         if not isinstance(meso_question_ids, list):
5728:             meso_question_ids = []
5729:
5730:         cluster_micros = [
5731:             m for m in micro_analyses
5732:             if m.question_id in meso_question_ids
5733:         ]
5734:         self.contradiction_scanner.scan_micro_meso(cluster_micros, meso)
5735:
5736:     # Calculate provisional macro score
5737:     if meso_analyses:
5738:         provisional_macro = np.mean([m.adjusted_score for m in meso_analyses])
5739:         self.contradiction_scanner.scan_meso_macro(meso_analyses, provisional_macro)
5740:
5741:     # Compose final macro portfolio
5742:     macro_analysis = self.portfolio_composer.compose_macro_portfolio(
5743:         meso_analyses,
5744:         total_questions,
5745:         self.contradiction_scanner
5746:     )
5747:
5748:     self.logger.info(f"  Detected {macro_analysis.contradiction_count} contradictions")
5749:     self.logger.info(f"  Final macro score: {macro_analysis.adjusted_score:.4f}")
5750:
5751:     return macro_analysis
5752:
5753: # =====
5754: # MAIN ENTRY POINT
5755: # =====
5756:
```

12/07/25

01:17:16

/Users/recovered/Applications/F.A.R.F.A.N -MECHANISTIC-PIPELINE/code_audit_pdfs/batch_05_combined.txt

106

5757: # Note: Main entry point removed to maintain I/O boundary separation.

5758: # For usage examples, see examples/ directory.

5759:

5760: