

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 30
3: =====
4: Generated: 2025-12-07T06:17:37.702282
5: Files in this batch: 11
6: =====
7:
8:
9: =====
10: FILE: tools/scan_boundaries.py
11: =====
12:
13: #!/usr/bin/env python3
14: """
15: AST Scanner for Core Module Boundary Violations
16:
17: Scans Python modules for:
18: 1. I/O operations (open, json.load/dump, pickle, pandas read_*, etc.)
19: 2. __main__ blocks
20: 3. Side effects on import
21: 4. subprocess, requests, click usage
22:
23: Usage:
24:     python tools/scan_boundaries.py --root src --fail-on=io,subprocess,requests,main
25:                                     --allow-path src/examples src/cli
26:                                     --sarif out/boundaries.sarif --json out/violations.json
27:
28: Exit code 0 if clean, 1 if violations found.
29: """
30:
31: import argparse
32: import ast
33: import json
34: import sys
35: from datetime import datetime
36: from pathlib import Path
37:
38: class BoundaryViolationVisitor(ast.NodeVisitor):
39:     """AST visitor to detect boundary violations in core modules."""
40:
41:     # I/O function names to detect
42:     IO_FUNCTIONS = {
43:         'open', 'read', 'write',
44:         'load', 'dump', 'loads', 'dumps',
45:         'read_csv', 'read_excel', 'read_json', 'read_sql', 'read_parquet',
46:         'to_csv', 'to_excel', 'to_json', 'to_sql', 'to_parquet',
47:         'read_text', 'write_text', 'read_bytes', 'write_bytes',
48:     }
49:
50:     # Module names that indicate I/O
51:     IO_MODULES = {
52:         'pickle', 'json', 'yaml', 'toml', 'csv',
53:     }
54:
55:     # Subprocess/network modules
56:     SUBPROCESS_MODULES = {'subprocess', 'os.system'}
```

```
57: NETWORK_MODULES = {'requests', 'urllib', 'http', 'httpx'}
58: CLI_MODULES = {'click', 'argparse', 'sys.argv'}
59:
60: def __init__(self, filename: str) -> None:
61:     self.filename = filename
62:     self.violations: list[dict[str, any]] = []
63:     self.has_main_block = False
64:     self.io_calls: list[tuple[int, str]] = []
65:     self.subprocess_calls: list[tuple[int, str]] = []
66:     self.network_calls: list[tuple[int, str]] = []
67:     self.cli_usage: list[tuple[int, str]] = []
68:
69: def visit_If(self, node: ast.If) -> None:
70:     """Detect if __name__ == '__main__' blocks."""
71:     # Check for __name__ == '__main__' pattern
72:     if isinstance(node.test, ast.Compare) and isinstance(node.test.left, ast.Name) and node.test.left.id == '__name__':
73:         for comparator in node.test.comparators:
74:             if isinstance(comparator, ast.Constant) and comparator.value == '__main__':
75:                 self.has_main_block = True
76:                 self.violations.append({
77:                     'type': 'main_block',
78:                     'line': node.lineno,
79:                     'message': f'__main__ block found at line {node.lineno}'
80:                 })
81:             self.generic_visit(node)
82:
83: def visit_Call(self, node: ast.Call) -> None:
84:     """Detect I/O function calls."""
85:     # Direct function calls
86:     if isinstance(node.func, ast.Name):
87:         if node.func.id in self.IO_FUNCTIONS:
88:             self.io_calls.append((node.lineno, node.func.id))
89:             self.violations.append({
90:                 'type': 'io_call',
91:                 'line': node.lineno,
92:                 'function': node.func.id,
93:                 'message': f'I/O operation {node.func.id}() at line {node.lineno}'
94:             })
95:
96:     # Module.function calls (e.g., json.load)
97:     elif isinstance(node.func, ast.Attribute) and isinstance(node.func.value, ast.Name):
98:         module_name = node.func.value.id
99:         func_name = node.func.attr
100:
101:         # Check for I/O
102:         if module_name in self.IO_MODULES or func_name in self.IO_FUNCTIONS:
103:             self.io_calls.append((node.lineno, f'{module_name}.{func_name}'))
104:             self.violations.append({
105:                 'type': 'io_call',
106:                 'line': node.lineno,
107:                 'function': f'{module_name}.{func_name}',
108:                 'message': f'I/O operation {module_name}.{func_name}() at line {node.lineno}'
109:             })
110:
111:         # Check for subprocess
112:         if module_name in self.SUBPROCESS_MODULES:
```

```
113:         self.subprocess_calls.append((node.lineno, f'{module_name}.{func_name}'))
114:         self.violations.append({
115:             'type': 'subprocess_call',
116:             'line': node.lineno,
117:             'function': f'{module_name}.{func_name}',
118:             'message': f'Subprocess call {module_name}.{func_name}() at line {node.lineno}'
119:         })
120:
121:         # Check for network
122:         if module_name in self.NETWORK_MODULES:
123:             self.network_calls.append((node.lineno, f'{module_name}.{func_name}'))
124:             self.violations.append({
125:                 'type': 'network_call',
126:                 'line': node.lineno,
127:                 'function': f'{module_name}.{func_name}',
128:                 'message': f'Network call {module_name}.{func_name}() at line {node.lineno}'
129:             })
130:
131:         self.generic_visit(node)
132:
133:     def visit_With(self, node: ast.With) -> None:
134:         """Detect 'with open(...)' patterns."""
135:         for item in node.items:
136:             if (isinstance(item.context_expr, ast.Call) and
137:                 isinstance(item.context_expr.func, ast.Name) and
138:                 item.context_expr.func.id == 'open'):
139:                 self.io_calls.append((node.lineno, 'open (with)'))
140:                 self.violations.append({
141:                     'type': 'io_call',
142:                     'line': node.lineno,
143:                     'function': 'open',
144:                     'message': f'I/O operation: with open(...) at line {node.lineno}'
145:                 })
146:         self.generic_visit(node)
147:
148:     def scan_file(filepath: Path) -> dict[str, any]:
149:         """Scan a single Python file for boundary violations."""
150:         try:
151:             with open(filepath, encoding='utf-8') as f:
152:                 source = f.read()
153:         except Exception as e:
154:             return {
155:                 'file': str(filepath),
156:                 'error': f'Could not read file: {e}',
157:                 'violations': [],
158:                 'clean': False
159:             }
160:
161:         try:
162:             tree = ast.parse(source, filename=str(filepath))
163:         except SyntaxError as e:
164:             return {
165:                 'file': str(filepath),
166:                 'error': f'Syntax error: {e}',
167:                 'violations': [],
168:                 'clean': False
```

```
169:     }
170:
171:     visitor = BoundaryViolationVisitor(str(filepath))
172:     visitor.visit(tree)
173:
174:     return {
175:         'file': str(filepath),
176:         'violations': visitor.violations,
177:         'has_main_block': visitor.has_main_block,
178:         'io_call_count': len(visitor.io_calls),
179:         'clean': len(visitor.violations) == 0,
180:         'error': None
181:     }
182:
183: def scan_directory(directory: Path, pattern: str = '*.py') -> list[dict[str, any]]:
184:     """Scan all Python files in a directory."""
185:     results = []
186:     for filepath in sorted(directory.rglob(pattern)):
187:         # Skip __pycache__ and test files
188:         if '__pycache__' in str(filepath) or 'test_' in filepath.name:
189:             continue
190:         results.append(scan_file(filepath))
191:     return results
192:
193: def generate_sarif_report(results: list[dict[str, any]], tool_version: str = "1.0.0") -> dict:
194:     """Generate SARIF 2.1.0 format report for GitHub annotations."""
195:     sarif = {
196:         "version": "2.1.0",
197:         "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-schema-2.1.0.json",
198:         "runs": [
199:             {
200:                 "tool": {
201:                     "driver": {
202:                         "name": "BoundaryScanner",
203:                         "version": tool_version,
204:                         "informationUri": "https://github.com/kkkkknhh/SAAAAAA",
205:                         "rules": [
206:                             {
207:                                 "id": "IO_VIOLATION",
208:                                 "name": "I/O Operation in Core Module",
209:                                 "shortDescription": {
210:                                     "text": "Core modules must not perform I/O operations"
211:                                 },
212:                                 "fullDescription": {
213:                                     "text": "All I/O operations must be performed through ports and adapters"
214:                                 },
215:                                 "defaultConfiguration": {
216:                                     "level": "error"
217:                                 }
218:                             },
219:                             {
220:                                 "id": "MAIN_BLOCK",
221:                                 "name": "__main__ Block in Core Module",
222:                                 "shortDescription": {
223:                                     "text": "Core modules must not contain __main__ blocks"
224:                                 },
```

```
225:         "defaultConfiguration": {
226:             "level": "error"
227:         }
228:     },
229:     {
230:         "id": "SUBPROCESS_VIOLATION",
231:         "name": "Subprocess Call in Core Module",
232:         "shortDescription": {
233:             "text": "Core modules must not call subprocess"
234:         },
235:         "defaultConfiguration": {
236:             "level": "error"
237:         }
238:     },
239:     {
240:         "id": "NETWORK_VIOLATION",
241:         "name": "Network Call in Core Module",
242:         "shortDescription": {
243:             "text": "Core modules must not make network calls"
244:         },
245:         "defaultConfiguration": {
246:             "level": "error"
247:         }
248:     }
249: ]
250: }
251: },
252: "results": []
253: }
254: ]
255: }
256:
257: for result in results:
258:     if not result['clean'] and not result.get('error'):
259:         for violation in result['violations']:
260:             rule_id = {
261:                 'io_call': 'IO_VIOLATION',
262:                 'main_block': 'MAIN_BLOCK',
263:                 'subprocess_call': 'SUBPROCESS_VIOLATION',
264:                 'network_call': 'NETWORK_VIOLATION'
265:             }.get(violation['type'], 'IO_VIOLATION')
266:
267:             sarif_result = {
268:                 "ruleId": rule_id,
269:                 "level": "error",
270:                 "message": {
271:                     "text": violation['message']
272:                 },
273:                 "locations": [
274:                     {
275:                         "physicalLocation": {
276:                             "artifactLocation": {
277:                                 "uri": result['file'],
278:                                 "uriBaseId": "%SRCROOT%"
279:                             },
280:                             "region": {
```

```
281:                 "startLine": violation['line'],
282:                 "startColumn": 1
283:             }
284:         }
285:     }
286: ]
287: }
288:     sarif['runs'][0]['results'].append(sarif_result)
289:
290: return sarif
291:
292: def generate_json_report(results: list[dict[str, any]]) -> dict:
293:     """Generate JSON violations report keyed by file, line, and node type."""
294:     violations_by_file = {}
295:
296:     for result in results:
297:         if not result['clean']:
298:             file_path = result['file']
299:             violations_by_file[file_path] = {
300:                 'violations': result['violations'],
301:                 'has_main_block': result.get('has_main_block', False),
302:                 'io_call_count': result.get('io_call_count', 0),
303:                 'error': result.get('error')
304:             }
305:
306:     return {
307:         'timestamp': datetime.now().isoformat(),
308:         'total_files_scanned': len(results),
309:         'files_with_violations': len(violations_by_file),
310:         'total_violations': sum(len(r['violations']) for r in results),
311:         'violations_by_file': violations_by_file
312:     }
313:
314: def should_allow_path(filepath: Path, allowed_paths: list[str]) -> bool:
315:     """Check if filepath is in any of the allowed paths."""
316:     filepath_str = str(filepath)
317:     return any(allowed in filepath_str for allowed in allowed_paths)
318:
319: def print_report(results: list[dict[str, any]], fail_on_types: set[str] | None = None) -> int:
320:     """Print scan results and return exit code."""
321:     total_files = len(results)
322:     clean_files = sum(1 for r in results if r['clean'])
323:     total_violations = sum(len(r['violations']) for r in results)
324:
325:     if fail_on_types is None:
326:         fail_on_types = {'io_call', 'main_block', 'subprocess_call', 'network_call'}
327:
328:     print("=" * 80)
329:     print("CORE MODULE BOUNDARY SCAN REPORT")
330:     print("=" * 80)
331:     print(f"\nFiles scanned: {total_files}")
332:     print(f"Clean files: {clean_files}")
333:     print(f"Files with violations: {total_files - clean_files}")
334:     print(f"Total violations: {total_violations}")
335:     print()
336:
```

```
337:     if total_violations == 0:
338:         print("â\234\205 All files are clean! No boundary violations detected.")
339:         return 0
340:
341:     print("â\235\214 Violations found:\n")
342:
343:     # Count violations by type
344:     violation_counts = {}
345:     for result in results:
346:         if not result['clean']:
347:             for violation in result['violations']:
348:                 vtype = violation['type']
349:                 violation_counts[vtype] = violation_counts.get(vtype, 0) + 1
350:
351:     print("Violation summary:")
352:     for vtype, count in sorted(violation_counts.items()):
353:         marker = "â\235\214" if vtype in fail_on_types else "â\232 ï,\217 "
354:         print(f" {marker} {vtype}: {count}")
355:     print()
356:
357:     for result in results:
358:         if not result['clean']:
359:             print(f"\n{result['file']}")
360:             if result.get('error'):
361:                 print(f" ERROR: {result['error']}")
362:             else:
363:                 for violation in result['violations']:
364:                     marker = "â\235\214" if violation['type'] in fail_on_types else "â\232 ï,\217 "
365:                     print(f" {marker} Line {violation['line']}: {violation['message']}")
366:
367:     print("\n" + "=" * 80)
368:     print("REMEDIATION:")
369:     print("- Move all __main__ blocks to examples/ directory")
370:     print("- Move all I/O operations to orchestrator/factory.py")
371:     print("- Core modules should be pure libraries receiving data via contracts")
372:     print("=" * 80)
373:
374:     # Determine if we should fail based on fail_on_types
375:     should_fail = any(
376:         violation['type'] in fail_on_types
377:         for result in results
378:         for violation in result['violations']
379:     )
380:
381:     return 1 if should_fail else 0
382:
383: def main() -> int:
384:     """Main entry point."""
385:     parser = argparse.ArgumentParser(
386:         description='Scan Python modules for boundary violations',
387:         formatter_class=argparse.RawDescriptionHelpFormatter,
388:         epilog="""
389: Examples:
390: # Basic scan
391: python tools/scan_boundaries.py --root src/farfan_pipeline/core
392:
```

```
393: # Fail on specific violations only
394: python tools/scan_boundaries.py --root src --fail-on io,main
395:
396: # Allow specific paths
397: python tools/scan_boundaries.py --root src --fail-on io,subprocess,requests,main \\
398:     --allow-path src/examples src/cli
399:
400: # Generate SARIF and JSON reports
401: python tools/scan_boundaries.py --root src/farfan_pipeline/core \\
402:     --sarif out/boundaries.sarif \\
403:     --json out/violations.json
404: """
405: )
406:
407: parser.add_argument(
408:     '--root',
409:     type=str,
410:     required=True,
411:     help='Root directory to scan'
412: )
413:
414: parser.add_argument(
415:     '--fail-on',
416:     type=str,
417:     default='io,main,subprocess,network',
418:     help='Comma-separated list of violation types to fail on (io, main, subprocess, network)'
419: )
420:
421: parser.add_argument(
422:     '--allow-path',
423:     nargs='+',
424:     default=[],
425:     help='Paths to exclude from scanning (e.g., src/examples src/cli)'
426: )
427:
428: parser.add_argument(
429:     '--sarif',
430:     type=str,
431:     help='Output SARIF report to this file'
432: )
433:
434: parser.add_argument(
435:     '--json',
436:     type=str,
437:     help='Output JSON violations report to this file'
438: )
439:
440: # Legacy positional argument support
441: if len(sys.argv) == 2 and not sys.argv[1].startswith('--'):
442:     # Old style: python scan_boundaries.py <directory>
443:     target_path = Path(sys.argv[1])
444:     fail_on_types = {'io_call', 'main_block', 'subprocess_call', 'network_call'}
445:     allowed_paths = []
446:     sarif_output = None
447:     json_output = None
448: else:
```



```
449:     args = parser.parse_args()
450:     target_path = Path(args.root)
451:     fail_on_types = set()
452:     for vtype in args.fail_on.split(','):
453:         vtype = vtype.strip()
454:         if vtype == 'io':
455:             fail_on_types.add('io_call')
456:         elif vtype == 'main':
457:             fail_on_types.add('main_block')
458:         elif vtype == 'subprocess':
459:             fail_on_types.add('subprocess_call')
460:         elif vtype == 'network':
461:             fail_on_types.add('network_call')
462:
463:     allowed_paths = args.allow_path
464:     sarif_output = args.sarif
465:     json_output = args.json
466:
467:     if not target_path.exists():
468:         print(f"Error: Directory {target_path} does not exist")
469:         return 1
470:
471:     results = scan_directory(target_path)
472:
473:     # Filter results based on allowed paths
474:     if allowed_paths:
475:         filtered_results = []
476:         for result in results:
477:             if not should_allow_path(Path(result['file']), allowed_paths):
478:                 filtered_results.append(result)
479:             else:
480:                 # Mark as clean if in allowed path
481:                 result['clean'] = True
482:                 result['violations'] = []
483:                 filtered_results.append(result)
484:         results = filtered_results
485:
486:     # Generate SARIF report if requested
487:     if sarif_output:
488:         sarif_path = Path(sarif_output)
489:         sarif_path.parent.mkdir(parents=True, exist_ok=True)
490:         sarif_data = generate_sarif_report(results)
491:         with open(sarif_path, 'w', encoding='utf-8') as f:
492:             json.dump(sarif_data, f, indent=2)
493:         print(f"SARIF report written to {sarif_output}")
494:
495:     # Generate JSON report if requested
496:     if json_output:
497:         json_path = Path(json_output)
498:         json_path.parent.mkdir(parents=True, exist_ok=True)
499:         json_data = generate_json_report(results)
500:         with open(json_path, 'w', encoding='utf-8') as f:
501:             json.dump(json_data, f, indent=2)
502:         print(f"JSON report written to {json_output}")
503:
504:     return print_report(results, fail_on_types)
```

```
505:
506: if __name__ == '__main__':
507:     sys.exit(main())
508:
509:
510:
511: =====
512: FILE: tools/scan_core_purity.py
513: =====
514:
515: #!/usr/bin/env python3
516: """
517: Core Purity Scanner - Ensures core modules follow functional purity principles.
518:
519: Checks:
520: 1. No I/O operations in core modules (print, open, file operations)
521: 2. No __main__ blocks in core modules
522: 3. No direct database or network calls
523: """
524:
525: import ast
526: import sys
527: from pathlib import Path
528: from typing import List, Tuple
529:
530: try:
531:     from farfan_pipeline.config.paths import PROJECT_ROOT
532: except Exception: # pragma: no cover - bootstrap fallback
533:     PROJECT_ROOT = Path(__file__).resolve().parents[1]
534:
535: # Directories that must maintain purity
536: CORE_PATHS = [
537:     "src/farfan_pipeline/core",
538: ]
539:
540: # Forbidden operations (allowing open for config loading, but not print/input)
541: FORBIDDEN_FUNCTIONS = {
542:     "print", "input",
543: }
544:
545: FORBIDDEN_IMPORTS = {
546:     "requests", "urllib", "socket", "sqlalchemy",
547: }
548:
549:
550: class PurityChecker(ast.NodeVisitor):
551:     """AST visitor to detect impure operations."""
552:
553:     def __init__(self, filepath: Path):
554:         self.filepath = filepath
555:         self.violations: List[Tuple[int, str]] = []
556:
557:     def visit_Call(self, node: ast.Call):
558:         """Check for forbidden function calls."""
559:         if isinstance(node.func, ast.Name):
560:             if node.func.id in FORBIDDEN_FUNCTIONS:
```

```
561:         self.violations.append(
562:             (node.lineno, f"Forbidden function: {node.func.id}")
563:         )
564:     self.generic_visit(node)
565:
566: def visit_If(self, node: ast.If):
567:     """Check for __main__ blocks."""
568:     if isinstance(node.test, ast.Compare):
569:         if isinstance(node.test.left, ast.Name):
570:             if node.test.left.id == "__name__":
571:                 self.violations.append(
572:                     (node.lineno, "Forbidden __main__ block in core module")
573:                 )
574:     self.generic_visit(node)
575:
576: def visit_Import(self, node: ast.Import):
577:     """Check for forbidden imports."""
578:     for alias in node.names:
579:         if any(forbidden in alias.name for forbidden in FORBIDDEN_IMPORTS):
580:             self.violations.append(
581:                 (node.lineno, f"Forbidden import: {alias.name}")
582:             )
583:     self.generic_visit(node)
584:
585: def visit_ImportFrom(self, node: ast.ImportFrom):
586:     """Check for forbidden imports."""
587:     if node.module:
588:         if any(forbidden in node.module for forbidden in FORBIDDEN_IMPORTS):
589:             self.violations.append(
590:                 (node.lineno, f"Forbidden import from: {node.module}")
591:             )
592:     self.generic_visit(node)
593:
594:
595: def check_file_purity(filepath: Path) -> List[Tuple[int, str]]:
596:     """Check a single file for purity violations."""
597:     try:
598:         with open(filepath, "r", encoding="utf-8") as f:
599:             tree = ast.parse(f.read(), filename=str(filepath))
600:
601:             checker = PurityChecker(filepath)
602:             checker.visit(tree)
603:             return checker.violations
604:     except SyntaxError as e:
605:         return [(e.lineno or 0, f"Syntax error: {e.msg}")]
606:     except Exception as e:
607:         return [(0, f"Error parsing file: {e}")]
608:
609:
610: def main() -> int:
611:     """Scan all core modules for purity violations."""
612:     repo_root = PROJECT_ROOT
613:     violations_found = False
614:
615:     for core_path_str in CORE_PATHS:
616:         core_path = repo_root / core_path_str
```

```
617:
618:     if not core_path.exists():
619:         print(f"â\232 i,\217 Core path not found: {core_path}")
620:         continue
621:
622:     print(f"Scanning {core_path_str}...")
623:
624:     for py_file in core_path.rglob("*.py"):
625:         violations = check_file_purity(py_file)
626:
627:         if violations:
628:             violations_found = True
629:             rel_path = py_file.relative_to(repo_root)
630:             print(f"â\235\214 {rel_path}")
631:             for lineno, msg in violations:
632:                 print(f" Line {lineno}: {msg}")
633:
634:     if violations_found:
635:         print("â\235\214 Core purity violations detected")
636:         return 1
637:     else:
638:         print("â\234\223 All core modules are pure")
639:         return 0
640:
641:
642: if __name__ == "__main__":
643:     sys.exit(main())
644:
645:
646:
647: =====
648: FILE: tools/signal_ecosystem_validator.py
649: =====
650:
651: #!/usr/bin/env python3
652: """
653: Signal Ecosystem Validator - Análisis basado en AST
654:
655: Este script analiza SOLO imports reales en tu código para determinar
656: qué archivos de signals están realmente en uso.
657:
658: NO hace suposiciones. NO adivina. Solo lee el AST.
659: """
660:
661: import ast
662: from pathlib import Path
663: from typing import Dict, Set, List
664: from dataclasses import dataclass, field
665:
666:
667: @dataclass
668: class ModuleAnalysis:
669:     """Resultado del análisis de un módulo."""
670:
671:     filepath: Path
672:     imports_from_signals: Set[str] = field(default_factory=set)
```

```
673:     imported_by: Set[str] = field(default_factory=set)
674:     is_dead_code: bool = False
675:     reason: str = ""
676:
677:
678: class SignalEcosystemValidator:
679:     """Validador del ecosistema de signals basado en AST."""
680:
681:     SIGNAL_MODULES = [
682:         'signal_aliasing',
683:         'signal_cache_invalidation',
684:         'signal_consumption',
685:         'signal_context_scoper',
686:         'signal_contract_validator',
687:         'signal_evidence_extractor',
688:         'signal_fallback_fusion',
689:         'signal_intelligence_layer',
690:         'signal_loader',
691:         'signal_quality_metrics',
692:         'signal_registry',
693:         'signal_resolution',
694:         'signal_semantic_expander',
695:         'signals',
696:     ]
697:
698:     def __init__(self, src_root: Path):
699:         """Inicializa validador con ruta raíz del código."""
700:         self.src_root = src_root
701:         self.orchestrator_dir = src_root / "core" / "orchestrator"
702:         self.results: Dict[str, ModuleAnalysis] = {}
703:
704:     def analyze_file(self, filepath: Path) -> ModuleAnalysis:
705:         """Analiza un archivo Python y extrae sus imports reales."""
706:         analysis = ModuleAnalysis(filepath=filepath)
707:
708:         try:
709:             with open(filepath, 'r', encoding='utf-8') as f:
710:                 tree = ast.parse(f.read(), filename=str(filepath))
711:
712:                 # Extraer imports
713:                 for node in ast.walk(tree):
714:                     if isinstance(node, ast.ImportFrom):
715:                         module = node.module or ''
716:
717:                         # ¿Importa desde signals?
718:                         for signal_mod in self.SIGNAL_MODULES:
719:                             if signal_mod in module:
720:                                 analysis.imports_from_signals.add(signal_mod)
721:
722:                     elif isinstance(node, ast.Import):
723:                         for alias in node.names:
724:                             name = alias.name
725:                             for signal_mod in self.SIGNAL_MODULES:
726:                                 if signal_mod in name:
727:                                     analysis.imports_from_signals.add(signal_mod)
728:
```

```
729:         except Exception as e:
730:             analysis.reason = f"Error parsing: {e}"
731:
732:         return analysis
733:
734:     def scan_ecosystem(self) -> Dict[str, ModuleAnalysis]:
735:         """Escanea todo el ecosistema de orchestrator."""
736:
737:         # 1. Analizar TODOS los archivos de signals primero
738:         print("\nð\237\223\202 Scanning signal modules...")
739:         for signal_mod in self.SIGNAL_MODULES:
740:             signal_file = self.orchestrator_dir / f"{signal_mod}.py"
741:             if signal_file.exists():
742:                 self.results[signal_mod] = self.analyze_file(signal_file)
743:                 print(f"  â\234\223 {signal_mod}.py")
744:
745:         # 2. Construir grafo de dependencias ENTRE signals
746:         print("\nð\237\224\227 Building signal-to-signal dependency graph...")
747:         for signal_mod, analysis in self.results.items():
748:             for imported_signal in analysis.imports_from_signals:
749:                 if imported_signal in self.results:
750:                     self.results[imported_signal].imported_by.add(f"{signal_mod}.py")
751:                     print(f"  {signal_mod} â\206\222 imports â\206\222 {imported_signal}")
752:
753:         # 3. Analizar archivos principales del orchestrator
754:         print("\nð\237\223\213 Scanning main orchestrator files...")
755:         main_files = [
756:             'evidence_assembler.py', 'evidence_registry.py', 'evidence_validator.py', 'executor_config.py', 'executor_profiler.py', 'executors.py',
'core.py',
757:             'chunk_router.py',
758:             'resource_integration.py', 'resource_manager.py', 'task_planner.py', 'executor_config.py',
759:             'factory.py',
760:             'base_executor_with_contract.py',
761:             'irrigation_synchronizer.py',
762:             'arg_router.py',
763:             'method_registry.py',
764:         ]
765:
766:         for main_file in main_files:
767:             main_path = self.orchestrator_dir / main_file
768:             if main_path.exists():
769:                 analysis = self.analyze_file(main_path)
770:
771:                 # Marcar quÃ© signals son importados por archivos principales
772:                 for signal_mod in analysis.imports_from_signals:
773:                     if signal_mod in self.results:
774:                         self.results[signal_mod].imported_by.add(main_file)
775:                         print(f"  {main_file} â\206\222 imports â\206\222 {signal_mod}")
776:
777:         # 4. Identificar ROOT nodes (importados por cÃ³digo principal)
778:         print("\nð\237\214³ Identifying root nodes...")
779:         root_nodes = set()
780:         for signal_mod, analysis in self.results.items():
781:             for importer in analysis.imported_by:
782:                 if not importer.startswith('signal_'):
783:                     root_nodes.add(signal_mod)
```

```
784:         print(f"  ROOT: {signal_mod} (used by {importer}) ")
785:
786:     # 5. Propagar vitalidad desde roots usando BFS
787:     print("\nâ\231»i,\217 Propagating liveness from roots...")
788:     alive = set(root_nodes)
789:     queue = list(root_nodes)
790:
791:     while queue:
792:         current = queue.pop(0)
793:         if current not in self.results:
794:             continue
795:
796:         # Marcar dependencias transitivas como vivas
797:         for dep in self.results[current].imports_from_signals:
798:             if dep not in alive and dep in self.results:
799:                 alive.add(dep)
800:                 queue.append(dep)
801:                 print(f"  {dep} is alive (transitively via {current})")
802:
803:     # 6. Marcar código muerto (no alcanzable desde roots)
804:     print("\nð\237\222\200 Marking dead code...")
805:     for signal_mod, analysis in self.results.items():
806:         if signal_mod not in alive:
807:             analysis.is_dead_code = True
808:             analysis.reason = "Not reachable from any main orchestrator file"
809:             print(f"  DEAD: {signal_mod}")
810:
811:     return self.results
812:
813: def generate_report(self) -> str:
814:     """Genera reporte de análisis."""
815:
816:     lines = []
817:     lines.append("=" * 80)
818:     lines.append("SIGNAL ECOSYSTEM ANALYSIS - COMPLETE GRAPH TRAVERSAL")
819:     lines.append("=" * 80)
820:     lines.append("")
821:
822:     # ROOT NODES: Importados directamente por código principal
823:     root_nodes = {}
824:     for mod, analysis in self.results.items():
825:         for importer in analysis.imported_by:
826:             if not importer.startswith('signal_'):
827:                 root_nodes[mod] = analysis
828:                 break
829:
830:     if root_nodes:
831:         lines.append("ð\237\214³ ROOT NODES (imported by main orchestrator code)")
832:         lines.append("-" * 80)
833:         for mod, analysis in sorted(root_nodes.items()):
834:             importers = [i for i in analysis.imported_by if not i.startswith('signal_')]
835:             lines.append(f"  {mod}.py")
836:             lines.append(f"    â\206\222 Used by: {' , '.join(sorted(importers))}")
837:             lines.append("")
838:
839:     # TRANSITIVE: Importados indirectamente (signal â\206\222 signal)
```

```
840:         transitive = {}
841:         for mod, analysis in self.results.items():
842:             if not analysis.is_dead_code and mod not in root_nodes:
843:                 transitive[mod] = analysis
844:
845:         if transitive:
846:             lines.append("ð\237\224\227 TRANSITIVE DEPENDENCIES (signal â\206\222 signal chain)")
847:             lines.append("-" * 80)
848:             for mod, analysis in sorted(transitive.items()):
849:                 signal_importers = [i for i in analysis.imported_by if i.startswith('signal_')]
850:                 lines.append(f"    {mod}.py")
851:                 lines.append(f"        â\206\222 Used by signals: {' ', ' '.join(sorted(signal_importers))}")
852:             lines.append("")
853:
854:         # DEAD CODE: No alcanzable desde roots
855:         dead = {k: v for k, v in self.results.items() if v.is_dead_code}
856:         if dead:
857:             lines.append("ð\237\222\200 DEAD CODE (not reachable from any main file)")
858:             lines.append("-" * 80)
859:             for mod in sorted(dead.keys()):
860:                 lines.append(f"    {mod}.py")
861:             lines.append("")
862:
863:         # Estadísticas
864:         lines.append("ð\237\223\212 STATISTICS")
865:         lines.append("-" * 80)
866:         lines.append(f"Total signal modules analyzed: {len(self.results)}")
867:         lines.append(f"Root nodes (directly used): {len(root_nodes)}")
868:         lines.append(f"Transitive dependencies: {len(transitive)}")
869:         lines.append(f"Dead code modules: {len(dead)}")
870:         lines.append(f"Dead code percentage: {100 * len(dead) / len(self.results):.1f}%")
871:         lines.append("")
872:
873:         # Recomendaciones
874:         lines.append("â\234\205 RECOMMENDATIONS")
875:         lines.append("-" * 80)
876:         lines.append("1. KEEP ALL: Root nodes + transitive dependencies")
877:         lines.append("2. REVIEW: Dead code modules - verify with git blame/history")
878:         lines.append("3. MOVE: Experimental/proposed code to /experiments or /proposals")
879:         lines.append("4. DELETE: Confirmed abandoned code after team review")
880:         lines.append("")
881:
882:         # Dependency chains
883:         lines.append("ð\237\223\210 LONGEST DEPENDENCY CHAINS")
884:         lines.append("-" * 80)
885:         chains = self._find_longest_chains()
886:         for i, chain in enumerate(chains[:5], 1):
887:             lines.append(f"{i}. {' â\206\222 ' .join(chain)}")
888:         lines.append("")
889:
890:         return "\n".join(lines)
891:
892: def _find_longest_chains(self) -> List[List[str]]:
893:     """Encuentra las cadenas de dependencias más largas."""
894:     chains = []
895:
```



```
896:         # Empezar desde cada root node
897:         for mod, analysis in self.results.items():
898:             has_non_signal_importer = any(
899:                 not i.startswith('signal_') for i in analysis.imported_by
900:             )
901:             if has_non_signal_importer:
902:                 chains.extend(self._dfs_chains(mod, []))
903:
904:         # Ordenar por longitud descendente
905:         chains.sort(key=len, reverse=True)
906:         return chains
907:
908: def _dfs_chains(self, node: str, current_chain: List[str]) -> List[List[str]]:
909:     """DFS para encontrar todas las cadenas desde un nodo."""
910:     if node in current_chain: # Evitar ciclos
911:         return [current_chain + [node]]
912:
913:     new_chain = current_chain + [node]
914:
915:     if node not in self.results:
916:         return [new_chain]
917:
918:     deps = self.results[node].imports_from_signals
919:     if not deps:
920:         return [new_chain]
921:
922:     all_chains = []
923:     for dep in deps:
924:         all_chains.extend(self._dfs_chains(dep, new_chain))
925:
926:     return all_chains if all_chains else [new_chain]
927:
928: def get_dependency_graph(self) -> str:
929:     """Genera grafo de dependencias en formato texto."""
930:
931:     lines = []
932:     lines.append("DEPENDENCY GRAPH")
933:     lines.append("-" * 80)
934:
935:     for mod, analysis in sorted(self.results.items()):
936:         if not analysis.is_dead_code:
937:             lines.append(f"\n{mod}")
938:             if analysis.imports_from_signals:
939:                 for dep in sorted(analysis.imports_from_signals):
940:                     lines.append(f"    \u224a\u224a\u200a\u224a\u2000 depends on: {dep}")
941:             if analysis.imported_by:
942:                 for user in sorted(analysis.imported_by):
943:                     lines.append(f"    \u206\u221 used by: {user}")
944:
945:     return "\n".join(lines)
946:
947:
948: def main():
949:     """Ejecuta an\u00e1lisis del ecosistema."""
950:
951:     # AJUSTA ESTA RUTA a tu proyecto
```

```
952:     src_root = Path("src/farfan_pipeline")
953:
954:     if not src_root.exists():
955:         print(f"â\235\214 ERROR: {src_root} no existe")
956:         print("Ajusta la ruta 'src_root' en main()")
957:         return
958:
959:     validator = SignalEcosystemValidator(src_root)
960:
961:     print("Escaneando ecosystem...")
962:     validator.scan_ecosystem()
963:
964:     print(validator.generate_report())
965:     print(validator.get_dependency_graph())
966:
967:     # Exportar resultados
968:     report_file = Path("signal_ecosystem_analysis.txt")
969:     with open(report_file, 'w', encoding='utf-8') as f:
970:         f.write(validator.generate_report())
971:         f.write("\n\n")
972:         f.write(validator.get_dependency_graph())
973:
974:     print(f"â\234\205 Reporte guardado en: {report_file}")
975:
976:
977: if __name__ == "__main__":
978:     main()
979:
980:
981:
982: =====
983: FILE: tools/testing/__init__.py
984: =====
985:
986: """Testing utilities."""
987:
988:
989:
990: =====
991: FILE: tools/testing/boot_check.py
992: =====
993:
994: #!/usr/bin/env python3
995: """
996: Boot check script for CI and pre-production environments.
997:
998: Validates that all modules load correctly, runtime validators initialize,
999: and the registry is complete without ClassNotFoundError.
1000:
1001: Usage:
1002:     python tools/testing/boot_check.py
1003:     python tools/testing/boot_check.py --verbose
1004: """
1005:
1006: import importlib
1007: import sys
```

```
1008: import traceback
1009: from pathlib import Path
1010:
1011: # Add project root to Python path
1012:
1013: # Modules to validate
1014: CORE_MODULES = [
1015:     "orchestrator",
1016:     "scoring",
1017:     "recommendation_engine",
1018:     "validation_engine",
1019:     "policy_processor",
1020:     "embedding_policy",
1021:     "semantic_chunking_policy",
1022: ]
1023:
1024: OPTIONAL_MODULES = [
1025:     "derek_beach",
1026:     "contradiction_deteccion",
1027:     "teoria_cambio",
1028:     "financiero_viabilidad_tablas",
1029:     "macro_prompts",
1030:     "micro_prompts",
1031: ]
1032:
1033: def check_module_import(module_name: str, verbose: bool = False) -> tuple[bool, str]:
1034:     """
1035:     Try to import a module and return success status.
1036:
1037:     Returns:
1038:         Tuple of (success, error_message)
1039:     """
1040:     try:
1041:         if verbose:
1042:             print(f" Importing {module_name}...", end=" ")
1043:
1044:             importlib.import_module(module_name)
1045:
1046:             if verbose:
1047:                 print("\234\223")
1048:
1049:             return True, ""
1050:     except ModuleNotFoundError as e:
1051:         error = f"Module not found: {e}"
1052:         if verbose:
1053:             print(f"\234\227 {error}")
1054:         return False, error
1055:     except ImportError as e:
1056:         error = f"Import error: {e}"
1057:         if verbose:
1058:             print(f"\234\227 {error}")
1059:         return False, error
1060:     except Exception as e:
1061:         error = f"Unexpected error: {e}"
1062:         if verbose:
1063:             print(f"\234\227 {error}")
```

```
1064:         if verbose:
1065:             traceback.print_exc()
1066:         return False, error
1067:
1068: def check_registry_validation(verbose: bool = False) -> tuple[bool, str]:
1069:     """
1070:     Validate that the orchestrator registry loads without ClassNotFoundError.
1071:
1072:     Returns:
1073:     Tuple of (success, error_message)
1074:     """
1075:     try:
1076:         if verbose:
1077:             print(" Validating orchestrator registry...", end=" ")
1078:
1079:         # Try to import and access the registry
1080:         from farfan_pipeline.core.orchestrator import registry
1081:
1082:         # Try to validate all classes (if method exists)
1083:         if hasattr(registry, 'validate_all_classes'):
1084:             registry.validate_all_classes()
1085:
1086:         if verbose:
1087:             print("\234\223")
1088:
1089:         return True, ""
1090:     except NameError as e:
1091:         if "ClassNotFoundError" in str(e) or "not defined" in str(e):
1092:             error = f"ClassNotFoundError in registry: {e}"
1093:             if verbose:
1094:                 print(f"\234\227 {error}")
1095:             return False, error
1096:         raise
1097:     except AttributeError as e:
1098:         # Module or registry doesn't exist - return as informational
1099:         if verbose:
1100:             print(f"\232 Registry validation not available: {e}")
1101:         return True, "" # Don't fail if registry validation not implemented
1102:     except Exception as e:
1103:         error = f"Registry validation error: {e}"
1104:         if verbose:
1105:             print(f"\234\227 {error}")
1106:             traceback.print_exc()
1107:         return False, error
1108:
1109: def check_runtime_validators(verbose: bool = False) -> tuple[bool, str]:
1110:     """
1111:     Validate that runtime validators initialize correctly.
1112:
1113:     Returns:
1114:     Tuple of (success, error_message)
1115:     """
1116:     try:
1117:         if verbose:
1118:             print(" Initializing runtime validators...", end=" ")
1119:
```

```
1120:     # Try to import and initialize validators
1121:     from farfan_pipeline.validation.validation_engine import RuntimeValidator
1122:
1123:     validator = RuntimeValidator()
1124:
1125:     # Try to run health check if available
1126:     if hasattr(validator, 'health_check'):
1127:         validator.health_check()
1128:
1129:     if verbose:
1130:         print("\234\223")
1131:
1132:     return True, ""
1133: except ImportError:
1134:     # validation_engine doesn't exist or RuntimeValidator not available
1135:     if verbose:
1136:         print("\232 Runtime validators not available (skipping)")
1137:     return True, "" # Don't fail if not implemented
1138: except Exception as e:
1139:     error = f"Runtime validator initialization error: {e}"
1140:     if verbose:
1141:         print(f"\234\227 {error}")
1142:         traceback.print_exc()
1143:     return False, error
1144:
1145: def run_boot_checks(verbose: bool = False) -> int:
1146:     """
1147:     Run all boot checks.
1148:
1149:     Returns:
1150:         Exit code (0 = success, 1 = failure)
1151:     """
1152:     print("=" * 60)
1153:     print("Boot Check - Module and Runtime Validation")
1154:     print("=" * 60)
1155:
1156:     all_passed = True
1157:     failed_checks = []
1158:
1159:     # Check core modules
1160:     print("\nChecking core modules:")
1161:     core_failed = []
1162:     for module in CORE_MODULES:
1163:         success, error = check_module_import(module, verbose)
1164:         if not success:
1165:             all_passed = False
1166:             core_failed.append(f"{module}: {error}")
1167:             failed_checks.append(f"Core module {module} failed to load")
1168:
1169:     if not verbose:
1170:         if core_failed:
1171:             print(f" \234\227 {len(core_failed)} core module(s) failed to load")
1172:             for failure in core_failed[:3]:
1173:                 print(f"    - {failure}")
1174:             if len(core_failed) > 3:
1175:                 print(f"    ... and {len(core_failed) - 3} more")
```

```
1176:         else:
1177:             print(f"  â\234\223 All {len(CORE_MODULES)} core modules loaded successfully")
1178:
1179:         # Check optional modules
1180:         print("\nChecking optional modules:")
1181:         optional_failed = []
1182:         for module in OPTIONAL_MODULES:
1183:             success, error = check_module_import(module, verbose)
1184:             if not success:
1185:                 optional_failed.append(f"{module}: {error}")
1186:             # Don't fail overall for optional modules
1187:
1188:         if not verbose:
1189:             loaded_count = len(OPTIONAL_MODULES) - len(optional_failed)
1190:             print(f"  â\234\223 {loaded_count}/{len(OPTIONAL_MODULES)} optional modules loaded")
1191:             if optional_failed:
1192:                 print(f"  â\232  {len(optional_failed)} optional module(s) not available")
1193:
1194:         # Check registry
1195:         print("\nChecking orchestrator registry:")
1196:         success, error = check_registry_validation(verbose)
1197:         if not success:
1198:             all_passed = False
1199:             failed_checks.append(f"Registry validation failed: {error}")
1200:         elif not verbose:
1201:             print("  â\234\223 Registry validation passed")
1202:
1203:         # Check runtime validators
1204:         print("\nChecking runtime validators:")
1205:         success, error = check_runtime_validators(verbose)
1206:         if not success:
1207:             all_passed = False
1208:             failed_checks.append(f"Runtime validator initialization failed: {error}")
1209:         elif not verbose:
1210:             print("  â\234\223 Runtime validators initialized successfully")
1211:
1212:         # Summary
1213:         print("\n" + "=" * 60)
1214:         if all_passed:
1215:             print("â\234\223 All boot checks PASSED")
1216:             print("=" * 60)
1217:             return 0
1218:         else:
1219:             print("â\234\227 Some boot checks FAILED")
1220:             print("\nFailed checks:")
1221:             for check in failed_checks:
1222:                 print(f"  - {check}")
1223:             print("=" * 60)
1224:             return 1
1225:
1226: def main() -> None:
1227:     verbose = "--verbose" in sys.argv or "-v" in sys.argv
1228:
1229:     exit_code = run_boot_checks(verbose)
1230:     sys.exit(exit_code)
1231:
```

```
1232: if __name__ == "__main__":
1233:     main()
1234:
1235:
1236:
1237: =====
1238: FILE: tools/validation/__init__.py
1239: =====
1240:
1241: """Validation tools."""
1242:
1243:
1244:
1245: =====
1246: FILE: tools/validation/validate_build_hygiene.py
1247: =====
1248:
1249: #!/usr/bin/env python3
1250: """
1251: Validation script for build hygiene checklist.
1252: Verifies that the repository follows all build hygiene requirements.
1253: """
1254:
1255: import re
1256: import sys
1257: from pathlib import Path
1258:
1259: def check_python_version_pin() -> bool:
1260:     """Check that Python version is pinned to 3.12.x."""
1261:     print("â\234\223 Checking Python version pin...")
1262:
1263:     # Check .python-version
1264:     python_version_file = Path(".python-version")
1265:     if not python_version_file.exists():
1266:         print(" â\234\227 Missing .python-version file")
1267:         return False
1268:
1269:     version = python_version_file.read_text().strip()
1270:     if not version.startswith("3.12"):
1271:         print(f" â\234\227 .python-version should be 3.12.x, got {version}")
1272:         return False
1273:
1274:     # Check pyproject.toml
1275:     pyproject = Path("pyproject.toml").read_text()
1276:     if 'requires-python = "~3.12.0"' not in pyproject:
1277:         print(" â\234\227 pyproject.toml should have requires-python = \"~3.12.0\"")
1278:         return False
1279:
1280:     if 'pythonVersion = "3.12"' not in pyproject:
1281:         print(" â\234\227 pyproject.toml should have pythonVersion = \"3.12\"")
1282:         return False
1283:
1284:     print(" â\234\223 Python version properly pinned to 3.12.x")
1285:     return True
1286:
1287: def check_pinned_dependencies() -> bool:
```

```
1288:     """Check that all dependencies are pinned to exact versions."""
1289:     print("â\234\223 Checking dependency pinning...")
1290:
1291:     requirements = Path("requirements.txt")
1292:     if not requirements.exists():
1293:         print(" â\234\227 Missing requirements.txt")
1294:         return False
1295:
1296:     constraints = Path("constraints.txt")
1297:     if not constraints.exists():
1298:         print(" â\234\227 Missing constraints.txt")
1299:         return False
1300:
1301:     # Check for wildcards or open ranges in requirements.txt
1302:     content = requirements.read_text()
1303:     lines = [line.strip() for line in content.split('\n')]
1304:     if line.strip() and not line.strip().startswith('#')]
1305:
1306:     bad_patterns = []
1307:     for line in lines:
1308:         # Check for wildcard or open ranges in version specifiers
1309:         # Look for these patterns after package name and optional extras
1310:         # Match patterns like: package>=1.0, package~=1.0, package>1.0, package<2.0, package*
1311:         if (re.search(r'(>|=|>|<|\*)', line) and
1312:             re.search(r'^\[.*(\[.*\])?\s*(>|=|>|<|\*)', line)):
1313:             # Further validate it's in version spec position, not in package name
1314:             # Package name format: name[extras]==version
1315:             bad_patterns.append(line)
1316:
1317:     if bad_patterns:
1318:         print(" â\234\227 Found wildcards or open ranges in requirements.txt:")
1319:         for pattern in bad_patterns:
1320:             print(f"    - {pattern}")
1321:         return False
1322:
1323:     print(f" â\234\223 All {len(lines)} dependencies properly pinned with exact versions")
1324:     print(" â\234\223 constraints.txt exists")
1325:     return True
1326:
1327: def check_directory_structure() -> bool:
1328:     """Check that required directories exist with __init__.py files."""
1329:     print("â\234\223 Checking directory structure...")
1330:
1331:     required_dirs = [
1332:         "orchestrator",
1333:         "executors",
1334:         "contracts",
1335:         "tests",
1336:         "tools",
1337:         "examples",
1338:         "src/farfan_pipeline/core",
1339:     ]
1340:
1341:     missing_dirs = []
1342:     missing_init = []
1343:
```



```
1344:     for dir_path in required_dirs:
1345:         path = Path(dir_path)
1346:         if not path.exists():
1347:             missing_dirs.append(dir_path)
1348:         elif not (path / "__init__.py").exists():
1349:             missing_init.append(dir_path)
1350:
1351:     if missing_dirs:
1352:         print("  â\234\227 Missing directories:")
1353:         for d in missing_dirs:
1354:             print(f"    - {d}")
1355:         return False
1356:
1357:     if missing_init:
1358:         print("  â\234\227 Missing __init__.py in:")
1359:         for d in missing_init:
1360:             print(f"    - {d}")
1361:         return False
1362:
1363:     print("  â\234\223 All required directories exist with __init__.py files")
1364:     return True
1365:
1366: def check_pythonpath_config() -> bool:
1367:     """Check that setup.py exists for proper PYTHONPATH configuration."""
1368:     print("â\234\223 Checking PYTHONPATH configuration...")
1369:
1370:     setup_py = Path("setup.py")
1371:     if not setup_py.exists():
1372:         print("  â\234\227 Missing setup.py for pip install -e .")
1373:         return False
1374:
1375:     content = setup_py.read_text()
1376:     if 'find_packages' not in content:
1377:         print("  â\234\227 setup.py should use find_packages()")
1378:         return False
1379:
1380:     print("  â\234\223 setup.py exists for editable installation")
1381:     return True
1382:
1383: def check_centralized_config() -> bool:
1384:     """Check that centralized configuration exists."""
1385:     print("â\234\223 Checking centralized configuration...")
1386:
1387:     settings = Path("orchestrator/settings.py")
1388:     if not settings.exists():
1389:         print("  â\234\227 Missing orchestrator/settings.py")
1390:         return False
1391:
1392:     env_example = Path(".env.example")
1393:     if not env_example.exists():
1394:         print("  â\234\227 Missing .env.example")
1395:         return False
1396:
1397:     gitignore = Path(".gitignore")
1398:     if gitignore.exists():
1399:         content = gitignore.read_text()
```

```
1400:         if ".env" not in content:
1401:             print("  â\234\227 .gitignore should exclude .env files")
1402:             return False
1403:
1404:     print("  â\234\223 Centralized config exists (orchestrator/settings.py, .env.example)")
1405:     print("  â\234\223 .env properly excluded in .gitignore")
1406:     return True
1407:
1408: def main() -> int:
1409:     """Run all validation checks."""
1410:     print("=" * 60)
1411:     print("Build Hygiene Validation")
1412:     print("=" * 60)
1413:     print()
1414:
1415:     checks = [
1416:         check_python_version_pin,
1417:         check_pinned_dependencies,
1418:         check_directory_structure,
1419:         check_pythonpath_config,
1420:         check_centralized_config,
1421:     ]
1422:
1423:     results = []
1424:     for check in checks:
1425:         try:
1426:             results.append(check())
1427:             print()
1428:         except Exception as e:
1429:             print(f"  â\234\227 Error running check: {e}")
1430:             results.append(False)
1431:             print()
1432:
1433:     print("=" * 60)
1434:     if all(results):
1435:         print("â\234\223 All build hygiene checks passed!")
1436:         print("=" * 60)
1437:         return 0
1438:     else:
1439:         failed = sum(1 for r in results if not r)
1440:         print(f"â\234\227 {failed} check(s) failed")
1441:         print("=" * 60)
1442:         return 1
1443:
1444: if __name__ == "__main__":
1445:     sys.exit(main())
1446:
1447:
1448:
1449: =====
1450: FILE: tools/validation/validate_scoring_parity.py
1451: =====
1452:
1453: #!/usr/bin/env python3
1454: """
1455: Validate scoring parity across modalities.
```

```
1456:
1457: This script ensures that:
1458: 1. Normalization formulas are correct for each modality
1459: 2. Quality thresholds are identical across all modalities
1460: 3. Boundary conditions produce correct quality levels
1461: 4. No modality has an unfair advantage at quality boundaries
1462:
1463: Usage:
1464:     python tools/validation/validate_scoring_parity.py
1465:     python tools/validation/validate_scoring_parity.py --verbose
1466: """
1467:
1468: import sys
1469:
1470: # Quality thresholds (must be identical across all modalities)
1471: QUALITY_THRESHOLDS = {
1472:     "EXCELENTE": 0.85,
1473:     "BUENO": 0.70,
1474:     "ACEPTABLE": 0.55,
1475:     "INSUFICIENTE": 0.00
1476: }
1477:
1478: # Modality score ranges
1479: MODALITY_RANGES = {
1480:     "TYPE_A": (0, 4),
1481:     "TYPE_B": (0, 3),
1482:     "TYPE_C": (0, 3),
1483:     "TYPE_D": (0, 3),
1484:     "TYPE_E": (0, 3),
1485:     "TYPE_F": (0, 3),
1486: }
1487:
1488: def normalize_score(raw_score: float, modality: str) -> float:
1489:     """Normalize a raw score to [0, 1] range based on modality."""
1490:     min_score, max_score = MODALITY_RANGES[modality]
1491:     if raw_score < min_score or raw_score > max_score:
1492:         raise ValueError(f"Score {raw_score} out of range for {modality}: [{min_score}, {max_score}]")
1493:     return (raw_score - min_score) / (max_score - min_score)
1494:
1495: def determine_quality_level(normalized_score: float) -> str:
1496:     """Determine quality level from normalized score."""
1497:     # Use small epsilon for floating point comparison
1498:     epsilon = 1e-9
1499:     if normalized_score >= QUALITY_THRESHOLDS["EXCELENTE"] - epsilon:
1500:         return "EXCELENTE"
1501:     elif normalized_score >= QUALITY_THRESHOLDS["BUENO"] - epsilon:
1502:         return "BUENO"
1503:     elif normalized_score >= QUALITY_THRESHOLDS["ACEPTABLE"] - epsilon:
1504:         return "ACEPTABLE"
1505:     else:
1506:         return "INSUFICIENTE"
1507:
1508: def test_normalization_formulas() -> bool:
1509:     """Test that normalization formulas are correct."""
1510:     print("Testing normalization formulas...")
1511:
```

```
1512:     test_cases = [
1513:         ("TYPE_A", 0, 0.0),
1514:         ("TYPE_A", 2, 0.5),
1515:         ("TYPE_A", 4, 1.0),
1516:         ("TYPE_B", 0, 0.0),
1517:         ("TYPE_B", 1.5, 0.5),
1518:         ("TYPE_B", 3, 1.0),
1519:         ("TYPE_C", 0, 0.0),
1520:         ("TYPE_C", 1.5, 0.5),
1521:         ("TYPE_C", 3, 1.0),
1522:     ]
1523:
1524:     passed = 0
1525:     failed = 0
1526:
1527:     for modality, raw, expected in test_cases:
1528:         actual = normalize_score(raw, modality)
1529:         if abs(actual - expected) < 0.001:
1530:             passed += 1
1531:             if "--verbose" in sys.argv:
1532:                 print(f"  \234\223 {modality}: {raw} \206\222 {actual:.3f} (expected {expected:.3f})")
1533:         else:
1534:             failed += 1
1535:             print(f"  \234\227 {modality}: {raw} \206\222 {actual:.3f} (expected {expected:.3f})")
1536:
1537:     print(f"  Passed: {passed}/{passed + failed}")
1538:     return failed == 0
1539:
1540: def test_parity_at_thresholds() -> bool:
1541:     """Test that all modalities produce the same quality level at threshold scores."""
1542:     print("\nTesting parity at quality thresholds...")
1543:
1544:     # Calculate equivalent raw scores for each threshold
1545:     test_cases = []
1546:     for quality, threshold in QUALITY_THRESHOLDS.items():
1547:         if quality == "INSUFICIENTE":
1548:             continue # Skip lower bound
1549:
1550:         for modality, (min_score, max_score) in MODALITY_RANGES.items():
1551:             raw_score = min_score + threshold * (max_score - min_score)
1552:             test_cases.append((modality, raw_score, quality))
1553:
1554:     passed = 0
1555:     failed = 0
1556:
1557:     for modality, raw_score, expected_quality in test_cases:
1558:         normalized = normalize_score(raw_score, modality)
1559:         actual_quality = determine_quality_level(normalized)
1560:
1561:         if actual_quality == expected_quality:
1562:             passed += 1
1563:             if "--verbose" in sys.argv:
1564:                 print(f"  \234\223 {modality} at {raw_score:.2f} \206\222 {actual_quality}")
1565:         else:
1566:             failed += 1
1567:             print(f"  \234\227 {modality} at {raw_score:.2f} \206\222 {actual_quality} (expected {expected_quality})")
```

```
1568:
1569:     print(f" Passed: {passed}/{passed + failed}")
1570:     return failed == 0
1571:
1572: def test_boundary_conditions() -> bool:
1573:     """Test boundary conditions (just above/below thresholds)."""
1574:     print("\nTesting boundary conditions...")
1575:
1576:     # Test scores just below and just above EXCELENTE threshold
1577:
1578:     test_cases = [
1579:         # Just below EXCELENTE (should be BUENO)
1580:         ("TYPE_A", 3.396, "BUENO"), # 3.396/4 = 0.849
1581:         ("TYPE_B", 2.547, "BUENO"), # 2.547/3 = 0.849
1582:
1583:         # Just at EXCELENTE threshold
1584:         ("TYPE_A", 3.4, "EXCELENTE"), # 3.4/4 = 0.85
1585:         ("TYPE_B", 2.55, "EXCELENTE"), # 2.55/3 = 0.85
1586:
1587:         # Just above EXCELENTE
1588:         ("TYPE_A", 3.404, "EXCELENTE"), # 3.404/4 = 0.851
1589:         ("TYPE_B", 2.553, "EXCELENTE"), # 2.553/3 = 0.851
1590:     ]
1591:
1592:     passed = 0
1593:     failed = 0
1594:
1595:     for modality, raw_score, expected_quality in test_cases:
1596:         normalized = normalize_score(raw_score, modality)
1597:         actual_quality = determine_quality_level(normalized)
1598:
1599:         if actual_quality == expected_quality:
1600:             passed += 1
1601:             if "--verbose" in sys.argv:
1602:                 print(f"  â\234\223 {modality}: {raw_score:.3f} (norm={normalized:.4f}) â\206\222 {actual_quality}")
1603:         else:
1604:             failed += 1
1605:             print(f"  â\234\227 {modality}: {raw_score:.3f} (norm={normalized:.4f}) â\206\222 {actual_quality} (expected {expected_quality})")
1606:
1607:     print(f" Passed: {passed}/{passed + failed}")
1608:     return failed == 0
1609:
1610: def test_no_unfair_advantage() -> bool:
1611:     """Test that no modality has an unfair advantage at boundaries."""
1612:     print("\nTesting for unfair advantages...")
1613:
1614:     # For each quality threshold, calculate the "difficulty" (raw score needed)
1615:     # relative to the maximum possible score
1616:     difficulties = {}
1617:
1618:     for quality, threshold in QUALITY_THRESHOLDS.items():
1619:         if quality == "INSUFICIENTE":
1620:             continue
1621:
1622:         difficulties[quality] = {}
1623:         for modality, (min_score, max_score) in MODALITY_RANGES.items():
```

```
1624:         raw_needed = min_score + threshold * (max_score - min_score)
1625:         relative_difficulty = raw_needed / max_score
1626:         difficulties[quality][modality] = relative_difficulty
1627:
1628:     passed = 0
1629:     failed = 0
1630:
1631:     for quality, modality_difficulties in difficulties.items():
1632:         # All modalities should have the same relative difficulty
1633:         values = list(modality_difficulties.values())
1634:         max_diff = max(values) - min(values)
1635:
1636:         if max_diff < 0.001: # Allow 0.1% variance
1637:             passed += 1
1638:             if "--verbose" in sys.argv:
1639:                 print(f"  â\234\223 {quality}: all modalities have equal difficulty (max diff: {max_diff:.6f})")
1640:         else:
1641:             failed += 1
1642:             print(f"  â\234\227 {quality}: modalities have unequal difficulty (max diff: {max_diff:.6f})")
1643:             for modality, diff in modality_difficulties.items():
1644:                 print(f"      {modality}: {diff:.6f}")
1645:
1646:     print(f" Passed: {passed}/{passed + failed}")
1647:     return failed == 0
1648:
1649: def main() -> int:
1650:     """Run all parity validation tests."""
1651:     print("=" * 60)
1652:     print("Scoring Parity Validation")
1653:     print("=" * 60)
1654:
1655:     all_passed = True
1656:
1657:     # Run all tests
1658:     all_passed &= test_normalization_formulas()
1659:     all_passed &= test_parity_at_thresholds()
1660:     all_passed &= test_boundary_conditions()
1661:     all_passed &= test_no_unfair_advantage()
1662:
1663:     print("\n" + "=" * 60)
1664:     if all_passed:
1665:         print("â\234\223 All parity validation tests PASSED")
1666:         print("=" * 60)
1667:         return 0
1668:     else:
1669:         print("â\234\227 Some parity validation tests FAILED")
1670:         print("=" * 60)
1671:         return 1
1672:
1673: if __name__ == "__main__":
1674:     sys.exit(main())
1675:
1676:
1677:
1678: =====
1679: FILE: update_executors_memory.py
```

```
1680: =====
1681:
1682: #!/usr/bin/env python3
1683: """
1684: Script to systematically add memory safety metrics to all executor return statements.
1685: """
1686: import re
1687:
1688: def update_executors_file(filepath: str) -> None:
1689:     with open(filepath, 'r', encoding='utf-8') as f:
1690:         content = f.read()
1691:
1692:     pattern = r'(\s*)"execution_metrics": \{(\s+("methods_count":.*?\n\s+"all_succeeded":.*?)\s+)\}'
1693:
1694:     def replacer(match):
1695:         indent = match.group(1)
1696:         metrics_content = match.group(2)
1697:
1698:         if "memory_safety" in metrics_content:
1699:             return match.group(0)
1700:
1701:         return (
1702:             f'{indent}"execution_metrics": {(\n'
1703:             f'{indent}    {metrics_content},\n'
1704:             f'{indent}    "memory_safety": self._get_memory_metrics_summary()\n'
1705:             f'{indent})}'
1706:         )
1707:
1708:     updated_content = re.sub(pattern, replacer, content)
1709:
1710:     with open(filepath, 'w', encoding='utf-8') as f:
1711:         f.write(updated_content)
1712:
1713:     print(f"Updated {filepath}")
1714:
1715: if __name__ == "__main__":
1716:     update_executors_file("src/farfan_pipeline/core/orchestrator/executors.py")
1717:
1718:
1719:
1720: =====
1721: FILE: verify_canonical_inventory.py
1722: =====
1723:
1724: #!/usr/bin/env python3
1725: """Verify the structure of generated canonical inventory files."""
1726:
1727: import json
1728: import sys
1729:
1730:
1731: def verify_canonical_inventory():
1732:     print("Verifying canonical_method_inventory.json...")
1733:     with open("canonical_method_inventory.json") as f:
1734:         data = json.load(f)
1735:
```

```
1736:     assert "methods" in data, "Missing 'methods' key"
1737:     assert "metadata" in data, "Missing 'metadata' key"
1738:
1739:     metadata = data["metadata"]
1740:     assert "total_methods" in metadata
1741:     assert "scan_timestamp" in metadata
1742:     assert "source_directory" in metadata
1743:
1744:     methods = data["methods"]
1745:     assert len(methods) > 0, "No methods found"
1746:
1747:     sample_method = next(iter(methods.values()))
1748:     required_keys = [
1749:         "canonical_name",
1750:         "file_path",
1751:         "line_number",
1752:         "class_name",
1753:         "role",
1754:         "is_executor",
1755:         "signature",
1756:     ]
1757:     for key in required_keys:
1758:         assert key in sample_method, f"Missing key '{key}' in method"
1759:
1760:     assert (
1761:         "parameters" in sample_method["signature"]
1762:     ), "Missing 'parameters' in signature"
1763:
1764:     print(f"    â\234\223 Valid structure with {len(methods)} methods")
1765:     return True
1766:
1767:
1768: def verify_statistics():
1769:     print("Verifying method_statistics.json...")
1770:     with open("method_statistics.json") as f:
1771:         data = json.load(f)
1772:
1773:     required_keys = [
1774:         "total_methods",
1775:         "total_executors",
1776:         "by_role",
1777:         "by_module",
1778:         "executor_distribution",
1779:     ]
1780:     for key in required_keys:
1781:         assert key in data, f"Missing key '{key}'"
1782:
1783:     assert data["total_methods"] > 0
1784:     assert data["total_executors"] >= 0
1785:     assert len(data["by_role"]) > 0
1786:     assert len(data["by_module"]) > 0
1787:
1788:     print("    â\234\223 Valid structure")
1789:     print(f"    Total methods: {data['total_methods']}")
1790:     print(f"    Total executors: {data['total_executors']}")
1791:     print(f"    Roles: {list(data['by_role'].keys())}")
```



```
1792:     return True
1793:
1794:
1795: def verify_excluded():
1796:     print("Verifying excluded_methods.json...")
1797:     with open("excluded_methods.json") as f:
1798:         data = json.load(f)
1799:
1800:     assert "excluded_methods" in data, "Missing 'excluded_methods' key"
1801:     assert "exclusion_reason" in data, "Missing 'exclusion_reason' key"
1802:     assert "total_excluded" in data, "Missing 'total_excluded' key"
1803:
1804:     assert data["exclusion_reason"] == "never calibrate"
1805:     assert data["total_excluded"] == len(data["excluded_methods"])
1806:
1807:     if len(data["excluded_methods"]) > 0:
1808:         sample = data["excluded_methods"][0]
1809:         required_keys = [
1810:             "canonical_id",
1811:             "reason",
1812:             "method_name",
1813:             "file_path",
1814:             "line_number",
1815:         ]
1816:         for key in required_keys:
1817:             assert key in sample, f"Missing key '{key}' in excluded method"
1818:
1819:     print(f"â\234\223 Valid structure with {data['total_excluded']} excluded methods")
1820:     return True
1821:
1822:
1823: def main():
1824:     try:
1825:         verify_canonical_inventory()
1826:         verify_statistics()
1827:         verify_excluded()
1828:         print("\nâ\234\205 All files verified successfully!")
1829:         return 0
1830:     except Exception as e:
1831:         print(f"\nâ\235\214 Verification failed: {e}", file=sys.stderr)
1832:         return 1
1833:
1834:
1835: if __name__ == "__main__":
1836:     sys.exit(main())
1837:
1838:
1839:
1840: =====
1841: FILE: verify_inventory.py
1842: =====
1843:
1844: #!/usr/bin/env python3
1845: """Standalone inventory verification script - does not require pytest"""
1846:
1847: import json
```

```
1848: import sys
1849: from pathlib import Path
1850: from typing import Any
1851:
1852:
1853: def load_inventory() -> dict[str, Any] | None:
1854:     """Load the inventory JSON file"""
1855:     inventory_path = Path("methods_inventory_raw.json")
1856:
1857:     if not inventory_path.exists():
1858:         print(f"ERROR: Inventory file not found: {inventory_path}", file=sys.stderr)
1859:         return None
1860:
1861:     with open(inventory_path, encoding="utf-8") as f:
1862:         return json.load(f)
1863:
1864:
1865: MINIMUM_METHOD_COUNT = 200
1866:
1867:
1868: def test_minimum_method_count(inventory: dict[str, Any]) -> tuple[bool, str]:
1869:     """Verify at least 200 methods in inventory"""
1870:     total = inventory["metadata"]["total_methods"]
1871:     if total < MINIMUM_METHOD_COUNT:
1872:         return False, f"Insufficient methods: {total} < {MINIMUM_METHOD_COUNT}"
1873:     return True, f"\234\223 Method count: {total} >= {MINIMUM_METHOD_COUNT}"
1874:
1875:
1876: def test_critical_files_present(inventory: dict[str, Any]) -> tuple[bool, str]:
1877:     """Verify methods from critical files are present"""
1878:     methods = inventory["methods"]
1879:     source_files = {m["source_file"] for m in methods}
1880:
1881:     critical_files = [
1882:         "derek_beach.py",
1883:         "aggregation.py",
1884:         "executors.py",
1885:         "executors_contract.py",
1886:     ]
1887:
1888:     errors = []
1889:     for critical_file in critical_files:
1890:         found = any(critical_file in sf for sf in source_files)
1891:         if not found:
1892:             errors.append(f"Critical file not found: {critical_file}")
1893:
1894:     if errors:
1895:         return False, "\n ".join(errors)
1896:     return True, f"\234\223 All {len(critical_files)} critical files present"
1897:
1898:
1899: def test_critical_method_patterns(inventory: dict[str, Any]) -> tuple[bool, str]:
1900:     """Verify critical method patterns are present"""
1901:     methods = inventory["methods"]
1902:     canonical_ids = {m["canonical_identifier"] for m in methods}
1903:
```

```
1904:     patterns_to_check = [
1905:         "derek_beach",
1906:         "aggregation",
1907:         "executors",
1908:     ]
1909:
1910:     errors = []
1911:     for pattern in patterns_to_check:
1912:         found = any(pattern in cid.lower() for cid in canonical_ids)
1913:         if not found:
1914:             errors.append(f"No methods found matching pattern: {pattern}")
1915:
1916:     if errors:
1917:         return False, "\n ".join(errors)
1918:     return True, f"â234223 All {len(patterns_to_check)} critical patterns found"
1919:
1920:
1921: def test_all_roles_present(inventory: dict[str, Any]) -> tuple[bool, str]:
1922:     """Verify all expected roles are present"""
1923:     stats = inventory["statistics"]["by_role"]
1924:
1925:     expected_roles = [
1926:         "ingest",
1927:         "processor",
1928:         "analyzer",
1929:         "extractor",
1930:         "score",
1931:         "utility",
1932:         "orchestrator",
1933:         "core",
1934:         "executor",
1935:     ]
1936:
1937:     errors = []
1938:     for role in expected_roles:
1939:         if role not in stats:
1940:             errors.append(f"Role not found: {role}")
1941:
1942:     if errors:
1943:         return False, "\n ".join(errors)
1944:     return True, f"â234223 All {len(expected_roles)} roles present"
1945:
1946:
1947: def test_calibration_flags_set(inventory: dict[str, Any]) -> tuple[bool, str]:
1948:     """Verify calibration flags are properly set"""
1949:     methods = inventory["methods"]
1950:
1951:     calibration_count = sum(1 for m in methods if m["requiere_calibracion"])
1952:     parametrization_count = sum(1 for m in methods if m["requiere_parametrizacion"])
1953:
1954:     if calibration_count == 0:
1955:         return False, "No methods flagged for calibration"
1956:     if parametrization_count == 0:
1957:         return False, "No methods flagged for parametrization"
1958:
1959:     return (
```

```
1960:         True,
1961:         f"â\234\223 Calibration: {calibration_count}, Parametrization: {parametrization_count}",
1962:     )
1963:
1964:
1965: MIN_CANONICAL_ID_PARTS = 2
1966: MAX_FORMAT_ERRORS = 5
1967:
1968:
1969: def test_canonical_identififer_format(inventory: dict[str, Any]) -> tuple[bool, str]:
1970:     """Verify canonical identifiers follow module.Class.method format"""
1971:     methods = inventory["methods"]
1972:     errors = []
1973:
1974:     for method in methods[:100]: # Sample first 100
1975:         cid = method["canonical_identififer"]
1976:         parts = cid.split(".")
1977:
1978:         if len(parts) < MIN_CANONICAL_ID_PARTS:
1979:             errors.append(f"Invalid canonical ID format: {cid}")
1980:             if len(errors) >= MAX_FORMAT_ERRORS:
1981:                 break
1982:
1983:     if errors:
1984:         return False, "\n ".join(errors)
1985:     return True, f"â\234\223 All canonical identifiers properly formatted"
1986:
1987:
1988: MIN_TAG_RATIO = 0.3
1989:
1990:
1991: def test_epistemology_tags_present(inventory: dict[str, Any]) -> tuple[bool, str]:
1992:     """Verify epistemology tags are assigned"""
1993:     methods = inventory["methods"]
1994:
1995:     tagged_count = sum(1 for m in methods if m["epistemology_tags"])
1996:     total = len(methods)
1997:
1998:     if tagged_count == 0:
1999:         return False, "No methods have epistemology tags"
2000:
2001:     tag_ratio = tagged_count / total
2002:     if tag_ratio < MIN_TAG_RATIO:
2003:         return False, f"Too few methods tagged: {tag_ratio:.2%}"
2004:
2005:     return True, f"â\234\223 Epistemology tags: {tagged_count}/{total} ({tag_ratio:.2%})"
2006:
2007:
2008: MIN_DEREK_BEACH_METHODS = 10
2009:
2010:
2011: def test_derek_beach_methods_complete(inventory: dict[str, Any]) -> tuple[bool, str]:
2012:     """Verify derek_beach.py methods are complete"""
2013:     methods = inventory["methods"]
2014:     derek_methods = [m for m in methods if "derek_beach" in m["source_file"]]
2015:
```

```
2016:     if len(derek_methods) < MIN_DEREK_BEACH_METHODS:
2017:         return False, f"Too few derek_beach methods: {len(derek_methods)}"
2018:
2019:     required_patterns = ["_format_message", "to_dict", "_load_config", "classify_test"]
2020:     errors = []
2021:
2022:     for pattern in required_patterns:
2023:         found = any(pattern in m["method_name"] for m in derek_methods)
2024:         if not found:
2025:             errors.append(f"Required derek_beach method not found: {pattern}")
2026:
2027:     if errors:
2028:         return False, "\n ".join(errors)
2029:     return (
2030:         True,
2031:         f"â\234\223 derek_beach.py: {len(derek_methods)} methods, all required patterns found",
2032:     )
2033:
2034:
2035: def test_aggregation_classes_present(inventory: dict[str, Any]) -> tuple[bool, str]:
2036:     """Verify aggregation classes are present"""
2037:     methods = inventory["methods"]
2038:     aggregation_methods = [
2039:         m for m in methods if "aggregation" in m["source_file"].lower()
2040:     ]
2041:
2042:     if len(aggregation_methods) == 0:
2043:         return False, "No aggregation methods found"
2044:
2045:     required_classes = [
2046:         "AreaPolicyAggregator",
2047:         "ClusterAggregator",
2048:         "DimensionAggregator",
2049:     ]
2050:
2051:     found_classes = {m["class_name"] for m in aggregation_methods if m["class_name"]}
2052:     errors = []
2053:
2054:     for req_class in required_classes:
2055:         if req_class not in found_classes:
2056:             errors.append(f"Required aggregation class not found: {req_class}")
2057:
2058:     if errors:
2059:         return False, "\n ".join(errors)
2060:     return (
2061:         True,
2062:         f"â\234\223 aggregation.py: {len(aggregation_methods)} methods, all classes found",
2063:     )
2064:
2065:
2066: MIN_EXECUTOR_METHODS = 5
2067:
2068:
2069: def test_executor_methods_present(inventory: dict[str, Any]) -> tuple[bool, str]:
2070:     """Verify executor methods are present"""
2071:     methods = inventory["methods"]
```

```
2072:     executor_methods = [m for m in methods if "executor" in m["source_file"].lower()]
2073:
2074:     if len(executor_methods) < MIN_EXECUTOR_METHODS:
2075:         return False, f"Too few executor methods: {len(executor_methods)}"
2076:
2077:     return True, f"â\234\223 executor files: {len(executor_methods)} methods"
2078:
2079:
2080: def test_no_duplicate_canonical_ids(inventory: dict[str, Any]) -> tuple[bool, str]:
2081:     """Verify no duplicate canonical identifiers"""
2082:     methods = inventory["methods"]
2083:     canonical_ids = [m["canonical_identifier"] for m in methods]
2084:
2085:     duplicates = [cid for cid in canonical_ids if canonical_ids.count(cid) > 1]
2086:
2087:     if duplicates:
2088:         unique_dupes = list(set(duplicates))[:5]
2089:         return False, f"Duplicate canonical IDs found: {unique_dupes}"
2090:
2091:     return True, "â\234\223 No duplicate canonical identifiers"
2092:
2093:
2094: def test_layer_requirements_complete(inventory: dict[str, Any]) -> tuple[bool, str]:
2095:     """Verify LAYER_REQUIREMENTS table is complete"""
2096:     layer_requirements = inventory["layer_requirements"]
2097:
2098:     expected_layers = [
2099:         "ingest",
2100:         "processor",
2101:         "analyzer",
2102:         "extractor",
2103:         "score",
2104:         "utility",
2105:         "orchestrator",
2106:         "core",
2107:         "executor",
2108:     ]
2109:
2110:     errors = []
2111:     for layer in expected_layers:
2112:         if layer not in layer_requirements:
2113:             errors.append(f"Layer missing from requirements: {layer}")
2114:         elif "description" not in layer_requirements[layer]:
2115:             errors.append(f"Layer missing description: {layer}")
2116:         elif "typical_patterns" not in layer_requirements[layer]:
2117:             errors.append(f"Layer missing typical_patterns: {layer}")
2118:
2119:     if errors:
2120:         return False, "\n ".join(errors)
2121:     return True, f"â\234\223 LAYER_REQUIREMENTS complete: {len(expected_layers)} layers"
2122:
2123:
2124: def main() -> None:
2125:     print("=" * 70)
2126:     print("INVENTORY COMPLETENESS VERIFICATION")
2127:     print("=" * 70)
```

```
2128: print()
2129:
2130: inventory = load_inventory()
2131: if inventory is None:
2132:     sys.exit(1)
2133:
2134: tests = [
2135:     ("Minimum method count", test_minimum_method_count),
2136:     ("Critical files present", test_critical_files_present),
2137:     ("Critical method patterns", test_critical_method_patterns),
2138:     ("All roles present", test_all_roles_present),
2139:     ("Calibration flags set", test_calibration_flags_set),
2140:     ("Canonical identifier format", test_canonical_identifier_format),
2141:     ("Epistemology tags present", test_epistemology_tags_present),
2142:     ("derek_beach methods complete", test_derek_beach_methods_complete),
2143:     ("aggregation classes present", test_aggregation_classes_present),
2144:     ("executor methods present", test_executor_methods_present),
2145:     ("No duplicate canonical IDs", test_no_duplicate_canonical_ids),
2146:     ("LAYER_REQUIREMENTS complete", test_layer_requirements_complete),
2147: ]
2148:
2149: passed = 0
2150: failed = 0
2151:
2152: for test_name, test_func in tests:
2153:     try:
2154:         success, message = test_func(inventory)
2155:         if success:
2156:             print(f"PASS: {test_name}")
2157:             print(f"      {message}")
2158:             passed += 1
2159:         else:
2160:             print(f"FAIL: {test_name}")
2161:             print(f"      {message}")
2162:             failed += 1
2163:     except Exception as e:
2164:         print(f"ERROR: {test_name}")
2165:         print(f"      {e}")
2166:         failed += 1
2167:     print()
2168:
2169: print("=" * 70)
2170: print(f"RESULTS: {passed} passed, {failed} failed")
2171: print("=" * 70)
2172:
2173: if failed > 0:
2174:     sys.exit(1)
2175: else:
2176:     print("\nâ\234\223â\234\223â\234\223 ALL TESTS PASSED â\234\223â\234\223â\234\223\n")
2177:     sys.exit(0)
2178:
2179:
2180: if __name__ == "__main__":
2181:     main()
2182:
2183:
```