

```
1: =====
2: F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 1
3: =====
4: Generated: 2025-12-07T06:17:14.414150
5: Files in this batch: 17
6: =====
7:
8:
9: =====
10: FILE: apply_fix.py
11: =====
12:
13: import os
14: import libcst as cst
15: from scripts.fix_farfan_imports import FixFarfanImports
16:
17: TARGET_DIR = "farfan_core"
18:
19: def apply_fix():
20:     print(f"Scanning {TARGET_DIR} for import fixes...")
21:     # Instantiate the transformer.
22:     transformer = FixFarfanImports()
23:
24:     count = 0
25:     for root, dirs, files in os.walk(TARGET_DIR):
26:         for file in files:
27:             if file.endswith(".py"):
28:                 path = os.path.join(root, file)
29:                 try:
30:                     with open(path, "r", encoding="utf-8") as f:
31:                         source = f.read()
32:
33:                         # Parse
34:                         tree = cst.parse_module(source)
35:
36:                         # Transform
37:                         modified_tree = tree.visit(transformer)
38:
39:                         # Check for changes
40:                         if modified_tree.code != source:
41:                             print(f"Fixing imports in: {path}")
42:                             with open(path, "w", encoding="utf-8") as f:
43:                                 f.write(modified_tree.code)
44:                             count += 1
45:
46:                         except Exception as e:
47:                             print(f"Failed to process {path}: {e}")
48:
49:     print(f"Refactoring complete. Modified {count} files.")
50:
51: if __name__ == "__main__":
52:     apply_fix()
53:
54:
55:
56: =====
```

```
57: FILE: artifacts/tmp/tmpxdn7ddfb/_remote_module_non_scriptable.py
58: =====
59:
60: from typing import *
61:
62: import torch
63: import torch.distributed.rpc as rpc
64: from torch import Tensor
65: from torch._jit_internal import Future
66: from torch.distributed.rpc import RRef
67: from typing import Tuple # pyre-ignore: unused import
68:
69:
70: module_interface_cls = None
71:
72:
73: def forward_async(self, *args, **kwargs):
74:     args = (self.module_rref, self.device, self.is_device_map_set, *args)
75:     kwargs = {**kwargs}
76:     return rpc.rpc_async(
77:         self.module_rref.owner(),
78:         _remote_forward,
79:         args,
80:         kwargs,
81:     )
82:
83:
84: def forward(self, *args, **kwargs):
85:     args = (self.module_rref, self.device, self.is_device_map_set, *args)
86:     kwargs = {**kwargs}
87:     ret_fut = rpc.rpc_async(
88:         self.module_rref.owner(),
89:         _remote_forward,
90:         args,
91:         kwargs,
92:     )
93:     return ret_fut.wait()
94:
95:
96: _generated_methods = [
97:     forward_async,
98:     forward,
99: ]
100:
101:
102:
103:
104: def _remote_forward(
105:     module_rref: RRef[module_interface_cls], device: str, is_device_map_set: bool, *args, **kwargs):
106:     module = module_rref.local_value()
107:     device = torch.device(device)
108:
109:     if device.type != "cuda":
110:         return module.forward(*args, **kwargs)
111:
112:     # If the module is on a cuda device,
```

```
113:     # move any CPU tensor in args or kwargs to the same cuda device.
114:     # Since torch script does not support generator expression,
115:     # have to use concatenation instead of
116:     # ``tuple(i.to(device) if isinstance(i, Tensor) else i for i in *args)``.
117:     args = (*args,)
118:     out_args: Tuple[()] = ()
119:     for arg in args:
120:         arg = (arg.to(device),) if isinstance(arg, Tensor) else (arg,)
121:         out_args = out_args + arg
122:
123:     kwargs = {**kwargs}
124:     for k, v in kwargs.items():
125:         if isinstance(v, Tensor):
126:             kwargs[k] = kwargs[k].to(device)
127:
128:     if is_device_map_set:
129:         return module.forward(*out_args, **kwargs)
130:
131:     # If the device map is empty, then only CPU tensors are allowed to send over wire,
132:     # so have to move any GPU tensor to CPU in the output.
133:     # Since torch script does not support generator expression,
134:     # have to use concatenation instead of
135:     # ``tuple(i.cpu() if isinstance(i, Tensor) else i for i in module.forward(*out_args, **kwargs))``.
136:     ret: Tuple[()] = ()
137:     for i in module.forward(*out_args, **kwargs):
138:         i = (i.cpu(),) if isinstance(i, Tensor) else (i,)
139:         ret = ret + i
140:     return ret
141:
142:
143:
144: =====
145: FILE: check_syntax.py
146: =====
147:
148: import os
149: import py_compile
150: import sys
151: import ast
152: from pathlib import Path
153: from typing import List, Tuple, Dict
154: from dataclasses import dataclass
155: import traceback
156:
157:
158: @dataclass
159: class SyntaxErrorRecord:
160:     file_path: str
161:     error_type: str
162:     line_number: int
163:     message: str
164:     full_traceback: str
165:
166:
167: class ComprehensiveSyntaxValidator:
168:     def __init__(self):
```

```
169:         self.errors: List[SyntaxErrorRecord] = []
170:         self.checked_files = 0
171:         self.directories_to_scan = [
172:             "src/farfan_pipeline",
173:             "farfan_core",
174:             "tests",
175:             "scripts",
176:             "tools",
177:         ]
178:
179:     def validate_file(self, file_path: str) -> None:
180:         """Validate a single Python file for syntax errors."""
181:         self.checked_files += 1
182:
183:         try:
184:             with open(file_path, 'r', encoding='utf-8') as f:
185:                 content = f.read()
186:
187:             try:
188:                 compile(content, file_path, 'exec')
189:             except SyntaxError as e:
190:                 error_msg = str(e)
191:                 line_num = e.lineno if e.lineno else 0
192:
193:                 error_detail = self._classify_error(e, content)
194:
195:                 self.errors.append(SyntaxErrorRecord(
196:                     file_path=file_path,
197:                     error_type=error_detail,
198:                     line_number=line_num,
199:                     message=error_msg,
200:                     full_traceback=traceback.format_exc()
201:                 ))
202:
203:             return
204:
205:             try:
206:                 ast.parse(content, filename=file_path)
207:             except SyntaxError as e:
208:                 error_msg = str(e)
209:                 line_num = e.lineno if e.lineno else 0
210:                 error_detail = self._classify_error(e, content)
211:
212:                 self.errors.append(SyntaxErrorRecord(
213:                     file_path=file_path,
214:                     error_type=error_detail,
215:                     line_number=line_num,
216:                     message=error_msg,
217:                     full_traceback=traceback.format_exc()
218:                 ))
219:
220:             try:
221:                 py_compile.compile(file_path, doraise=True)
222:             except py_compile.PyCompileError as e:
223:                 self.errors.append(SyntaxErrorRecord(
224:                     file_path=file_path,
```

```
225:             error_type="PyCompileError",
226:             line_number=0,
227:             message=str(e),
228:             full_traceback=traceback.format_exc()
229:         ))
230:
231:     except UnicodeDecodeError as e:
232:         self.errors.append(SyntaxErrorRecord(
233:             file_path=file_path,
234:             error_type="UnicodeDecodeError",
235:             line_number=0,
236:             message=f"Invalid encoding: {str(e)}",
237:             full_traceback=traceback.format_exc()
238:         ))
239:     except Exception as e:
240:         self.errors.append(SyntaxErrorRecord(
241:             file_path=file_path,
242:             error_type=type(e).__name__,
243:             line_number=0,
244:             message=str(e),
245:             full_traceback=traceback.format_exc()
246:         ))
247:
248: def _classify_error(self, error: Exception, content: str) -> str:
249:     """Classify the type of syntax error."""
250:     error_msg = str(error).lower()
251:
252:     if "missing parentheses" in error_msg or "unclosed" in error_msg:
253:         return "Unclosed brackets/parentheses"
254:     elif "invalid character" in error_msg or "non-ascii" in error_msg:
255:         return "Illegal characters"
256:     elif "unexpected indent" in error_msg or "expected an indented block" in error_msg:
257:         return "Invalid indentation"
258:     elif "invalid syntax" in error_msg and "f-string" in error_msg:
259:         return "Malformed f-strings"
260:     elif ":" in error_msg and "expected" in error_msg:
261:         return "Missing colons"
262:     elif "positional" in error_msg or "keyword" in error_msg:
263:         return "Python 3.10+ syntax violations"
264:     elif "invalid decimal literal" in error_msg:
265:         return "Invalid decimal literal"
266:     elif "from __future__" in error_msg:
267:         return "Invalid __future__ import placement"
268:     else:
269:         return "Syntax error"
270:
271: def scan_directory(self, directory: str) -> None:
272:     """Recursively scan a directory for Python files."""
273:     if not os.path.exists(directory):
274:         print(f"Warning: Directory {directory} does not exist")
275:     return
276:
277:     for root, dirs, files in os.walk(directory):
278:         dirs[:] = [d for d in dirs if d not in ['.git', '__pycache__', 'farfan-env', '.venv', 'venv']]
279:
280:         for file in files:
```

```
281:             if file.endswith('.py'):
282:                 file_path = os.path.join(root, file)
283:                 self.validate_file(file_path)
284:
285:     def generate_report(self) -> str:
286:         """Generate a markdown report of all syntax errors."""
287:         report = []
288:         report.append("# Python Syntax Validation Report\n")
289:         report.append(f"**Total files checked:** {self.checked_files}\n")
290:         report.append(f"**Files with errors:** {len(self.errors)}\n")
291:         report.append(f"**Success rate:** {((self.checked_files - len(self.errors)) / self.checked_files * 100):.2f}%\n\n")
292:
293:         if not self.errors:
294:             report.append("All Python files passed syntax validation!\n")
295:         return ''.join(report)
296:
297:         report.append("## Error Summary by Type\n\n")
298:         error_types: Dict[str, int] = {}
299:         for error in self.errors:
300:             error_types[error.error_type] = error_types.get(error.error_type, 0) + 1
301:
302:         for error_type, count in sorted(error_types.items(), key=lambda x: x[1], reverse=True):
303:             report.append(f"- **{error_type}**: {count} file(s)\n")
304:
305:         report.append("\n## Detailed Error List\n\n")
306:
307:         for idx, error in enumerate(self.errors, 1):
308:             report.append(f"### Error {idx}: {error.file_path}\n")
309:             report.append(f"- **Type:** {error.error_type}\n")
310:             report.append(f"- **Line:** {error.line_number}\n")
311:             report.append(f"- **Message:** {error.message}\n\n")
312:             report.append("```python\n")
313:             report.append(error.full_traceback)
314:             report.append("\n```\n")
315:
316:         return ''.join(report)
317:
318:     def run(self) -> int:
319:         """Run the validation process."""
320:         print("=" * 80)
321:         print("COMPREHENSIVE PYTHON SYNTAX VALIDATION")
322:         print("=" * 80)
323:         print()
324:
325:         for directory in self.directories_to_scan:
326:             print(f"Scanning directory: {directory}")
327:             self.scan_directory(directory)
328:
329:             print(f"\nValidation complete!")
330:             print(f"Total files checked: {self.checked_files}")
331:             print(f"Files with errors: {len(self.errors)}")
332:
333:             report_content = self.generate_report()
334:
335:             with open("SYNTAX_VALIDATION_REPORT.md", "w", encoding="utf-8") as f:
336:                 f.write(report_content)
```

```
337:
338:     print(f"\nReport saved to: SYNTAX_VALIDATION_REPORT.md")
339:
340:     if self.errors:
341:         print("\u2635\ufe0f VALIDATION FAILED - Syntax errors found")
342:         return 1
343:
344:     else:
345:         print("\u2634\ufe0f VALIDATION PASSED - No syntax errors found")
346:
347:
348: if __name__ == "__main__":
349:     validator = ComprehensiveSyntaxValidator()
350:     sys.exit(validator.run())
351:
352:
353:
354: =====
355: FILE: contracts/__init__.py
356: =====
357:
358: """
359: Legacy compatibility shim for the deprecated top-level 'contracts' package.
360:
361: Historically, downstream tooling imported 'contracts' from the repository root.
362: The canonical contract definitions now live in 'farfan_core.core.contracts'.
363: This shim keeps those imports functional and ensures that architectural tooling
364: placed under the 'contracts/' directory (such as importlinter configs) does not
365: shadow or break runtime behavior.
366:
367: DEPRECATED: This module is deprecated. Import from farfan_core.core.contracts instead.
368: """
369:
370: import warnings
371:
372: from farfan_core.core.contracts import * # noqa: F401,F403
373:
374: warnings.warn(
375:     "Importing from top-level 'contracts' package is deprecated. "
376:     "Use 'from farfan_core.core.contracts import ...' instead. "
377:     "This shim will be removed in a future version.",
378:     DeprecationWarning,
379:     stacklevel=2
380: )
381:
382:
383:
384: =====
385: FILE: debug_walk.py
386: =====
387:
388: from pathlib import Path
389:
390: from farfan_pipeline.core.method_inventory import walk_python_files, INVENTORY_ROOTS, _normalize_roots
391:
392: import os
```

```
393: print(f"Current CWD: {os.getcwd()}")
394: print(f"INVENTORY_ROOTS: {INVENTORY_ROOTS}")
395:
396: # Check if roots exist
397: for r in INVENTORY_ROOTS:
398:     print(f"Root {r} exists: {r.exists()}")
399:
400: files = walk_python_files()
401: print(f"Found {len(files)} files.")
402:
403: flux_count = 0
404: for f in files:
405:     if "flux" in str(f):
406:         print(f"Flux file: {f}")
407:         flux_count += 1
408:
409: print(f"Total flux files found: {flux_count}")
410:
411:
412:
413: =====
414: FILE: farfan_core/farfan_core/models/__init__.py
415: =====
416:
417: from farfan_core.farfan_core.models.execution_plan import (
418:     REQUIRED_TASK_COUNT,
419:     ExecutableTask,
420:     ExecutionPlan,
421: )
422: from farfan_core.farfan_core.models.micro_question_context import (
423:     MicroQuestionContext,
424:     sort_micro_question_contexts,
425: )
426:
427: __all__ = [
428:     "REQUIRED_TASK_COUNT",
429:     "ExecutableTask",
430:     "ExecutionPlan",
431:     "MicroQuestionContext",
432:     "sort_micro_question_contexts",
433: ]
434:
435:
436:
437: =====
438: FILE: farfan_core/farfan_core/models/execution_plan.py
439: =====
440:
441: """
442: Core immutable data models for execution plan management.
443:
444: Provides frozen dataclasses for task and execution plan representation with
445: deterministic integrity verification and strict immutability guarantees.
446: """
447:
448: from __future__ import annotations
```

```
449:
450: import hashlib
451: import json
452: import logging
453: from dataclasses import dataclass, field
454: from datetime import datetime, timezone
455: from pathlib import Path
456: from typing import Any
457:
458: REQUIRED_TASK_COUNT = 300
459:
460: logger = logging.getLogger(__name__)
461:
462:
463: @dataclass(frozen=True)
464: class ExecutableTask:
465:     """
466:         Immutable task representation for execution planning.
467:
468:         Enforces immutability through frozen=True and tuple-based collections
469:         to ensure deterministic execution and provenance tracking.
470:     """
471:
472:     task_id: str
473:     micro_question_context: str
474:     target_chunk: str
475:     applicable_patterns: tuple[str, ...]
476:     resolved_signals: tuple[str, ...]
477:     creation_timestamp: float
478:     synchronizer_version: str
479:
480:
481: @dataclass(frozen=True)
482: class ExecutionPlan:
483:     """
484:         Immutable execution plan containing exactly 300 tasks.
485:
486:         Enforces the 300-question analysis contract (D1-D6 dimensions \227 PA01-PA10 areas)
487:         with duplicate detection and cryptographic integrity verification.
488:     """
489:
490:     plan_id: str
491:     tasks: tuple[ExecutableTask, ...]
492:     metadata: dict[str, Any] = field(default_factory=dict) # type: ignore[misc]
493:
494:     def __post_init__(self) -> None:
495:         if len(self.tasks) != REQUIRED_TASK_COUNT:
496:             raise ValueError(
497:                 f"ExecutionPlan requires exactly {REQUIRED_TASK_COUNT} tasks, got {len(self.tasks)}")
498:
499:
500:     task_ids = [task.task_id for task in self.tasks]
501:     if len(task_ids) != len(set(task_ids)):
502:         seen: set[str] = set()
503:         duplicates: set[str] = set()
504:         for task_id in task_ids:
```

```
505:             if task_id in seen:
506:                 duplicates.add(task_id)
507:                 seen.add(task_id)
508:             raise ValueError(
509:                 f"ExecutionPlan contains duplicate task_ids: {sorted(duplicates)}"
510:             )
511:
512:     def compute_integrity_hash(self) -> str:
513:         """
514:             Compute deterministic SHA256 hash of serialized task list.
515:
516:             Returns:
517:                 Hexadecimal SHA256 hash string of JSON-serialized tasks.
518:
519:         task_data = [
520:             {
521:                 "task_id": task.task_id,
522:                 "micro_question_context": task.micro_question_context,
523:                 "target_chunk": task.target_chunk,
524:                 "applicable_patterns": list(task.applicable_patterns),
525:                 "resolved_signals": list(task.resolved_signals),
526:                 "creation_timestamp": task.creation_timestamp,
527:                 "synchronizer_version": task.synchronizer_version,
528:             }
529:             for task in self.tasks
530:         ]
531:
532:         serialized = json.dumps(task_data, sort_keys=True, separators=(", ", ":"))
```

```
561:     """
562:     Archive execution plan with atomic index update and transactional rollback.
563:
564:     Implements atomic write operation with three-phase commit:
565:     1. Write plan file to storage
566:     2. Append index entry to JSON Lines file
567:     3. Verify file integrity
568:
569:     Rollback triggers:
570:     - If index write fails: delete plan file via storage_path.unlink()
571:     - If file verification fails: remove index entry
572:
573:     Args:
574:         plan: ExecutionPlan instance to archive
575:         correlation_id: Optional correlation ID for tracking
576:
577:     Returns:
578:         Unmodified ExecutionPlan instance for immediate execution or disposal
579:
580:     Raises:
581:         ValueError: If plan archival fails
582:         OSError: If file system operations fail
583:     """
584:     created_at = datetime.now(timezone.utc).isoformat()
585:     task_count = len(plan.tasks)
586:     integrity_hash = plan.compute_integrity_hash()
587:     correlation_id_value = correlation_id or "unknown"
588:
589:     storage_filename = f"{plan.plan_id}_{created_at.replace(':', '-)}.json"
590:     storage_path = self.storage_dir / storage_filename
591:
592:     try:
593:         plan_data = {
594:             "plan_id": plan.plan_id,
595:             "tasks": [
596:                 {
597:                     "task_id": task.task_id,
598:                     "micro_question_context": task.micro_question_context,
599:                     "target_chunk": task.target_chunk,
600:                     "applicable_patterns": list(task.applicable_patterns),
601:                     "resolved_signals": list(task.resolved_signals),
602:                     "creation_timestamp": task.creation_timestamp,
603:                     "synchronizer_version": task.synchronizer_version,
604:                 }
605:                 for task in plan.tasks
606:             ],
607:             "metadata": plan.metadata,
608:             "created_at": created_at,
609:             "integrity_hash": integrity_hash,
610:             "correlation_id": correlation_id_value,
611:         }
612:
613:         storage_path.write_text(json.dumps(plan_data, indent=2), encoding="utf-8")
614:
615:     except OSError as e:
616:         logger.error(
```

```
617:             json.dumps(
618:                 {
619:                     "event": "execution_plan_archive_failed",
620:                     "plan_id": plan.plan_id,
621:                     "error": str(e),
622:                     "phase": "plan_file_write",
623:                     "correlation_id": correlation_id_value,
624:                 }
625:             )
626:         )
627:         raise ValueError(
628:             f"Failed to write plan file for {plan.plan_id}: {e}"
629:         ) from e
630:
631:     index_entry = {
632:         "plan_id": plan.plan_id,
633:         "storage_path": str(storage_path),
634:         "created_at": created_at,
635:         "task_count": task_count,
636:         "integrity_hash": integrity_hash,
637:         "correlation_id": correlation_id_value,
638:     }
639:
640:     try:
641:         with self.index_file.open("a", encoding="utf-8") as f:
642:             f.write(json.dumps(index_entry) + "\n")
643:
644:     except OSError as e:
645:         try:
646:             storage_path.unlink()
647:             logger.warning(
648:                 json.dumps(
649:                     {
650:                         "event": "execution_plan_archive_rollback",
651:                         "plan_id": plan.plan_id,
652:                         "reason": "index_write_failed",
653:                         "rollback_action": "plan_file_deleted",
654:                         "correlation_id": correlation_id_value,
655:                     }
656:                 )
657:             )
658:         except OSError as unlink_error:
659:             logger.error(
660:                 json.dumps(
661:                     {
662:                         "event": "execution_plan_archive_rollback_failed",
663:                         "plan_id": plan.plan_id,
664:                         "error": str(unlink_error),
665:                         "correlation_id": correlation_id_value,
666:                     }
667:                 )
668:             )
669:
670:         logger.error(
671:             json.dumps(
672:                 {
```

```
673:             "event": "execution_plan_archive_failed",
674:             "plan_id": plan.plan_id,
675:             "error": str(e),
676:             "phase": "index_write",
677:             "correlation_id": correlation_id_value,
678:         }
679:     )
680: )
681: raise ValueError(
682:     f"Failed to write index entry for {plan.plan_id}: {e}"
683: ) from e
684:
685: try:
686:     if not storage_path.exists():
687:         raise ValueError(f"Plan file verification failed: {storage_path}")
688:
689:     file_size = storage_path.stat().st_size
690:     if file_size == 0:
691:         raise ValueError(f"Plan file is empty: {storage_path}")
692:
693: except (OSError, ValueError) as e:
694:     try:
695:         self._remove_index_entry(plan.plan_id)
696:         logger.warning(
697:             json.dumps(
698:                 {
699:                     "event": "execution_plan_archive_rollback",
700:                     "plan_id": plan.plan_id,
701:                     "reason": "file_verification_failed",
702:                     "rollback_action": "index_entry_removed",
703:                     "correlation_id": correlation_id_value,
704:                 }
705:             )
706:         )
707:     except Exception as rollback_error:
708:         logger.error(
709:             json.dumps(
710:                 {
711:                     "event": "execution_plan_archive_rollback_failed",
712:                     "plan_id": plan.plan_id,
713:                     "error": str(rollback_error),
714:                     "correlation_id": correlation_id_value,
715:                 }
716:             )
717:         )
718:
719:         logger.error(
720:             json.dumps(
721:                 {
722:                     "event": "execution_plan_archive_failed",
723:                     "plan_id": plan.plan_id,
724:                     "error": str(e),
725:                     "phase": "file_verification",
726:                     "correlation_id": correlation_id_value,
727:                 }
728:             )
729:         )
730:     )
731: 
```

```
729:         )
730:         raise ValueError(f"File verification failed for {plan.plan_id}: {e}") from e
731:
732:     logger.info(
733:         json.dumps(
734:             {
735:                 "event": "execution_plan_archived",
736:                 "plan_id": plan.plan_id,
737:                 "storage_path": str(storage_path),
738:                 "created_at": created_at,
739:                 "task_count": task_count,
740:                 "integrity_hash": integrity_hash,
741:                 "correlation_id": correlation_id_value,
742:             }
743:         )
744:     )
745:
746:     return plan
747:
748: def _remove_index_entry(self, plan_id: str) -> None:
749:     """
750:     Remove index entry for given plan_id.
751:
752:     Rewrites index file excluding the entry with matching plan_id.
753:
754:     Args:
755:         plan_id: Plan ID to remove from index
756:     """
757:     if not self.index_file.exists():
758:         return
759:
760:     lines = []
761:     try:
762:         with self.index_file.open("r", encoding="utf-8") as f:
763:             lines = f.readlines()
764:     except OSError as e:
765:         logger.error(f"Failed to read index file for removal: {e}")
766:         raise
767:
768:     filtered_lines = []
769:     for line in lines:
770:         try:
771:             entry = json.loads(line.strip())
772:             if entry.get("plan_id") != plan_id:
773:                 filtered_lines.append(line)
774:         except json.JSONDecodeError:
775:             filtered_lines.append(line)
776:
777:     try:
778:         with self.index_file.open("w", encoding="utf-8") as f:
779:             f.writelines(filtered_lines)
780:     except OSError as e:
781:         logger.error(f"Failed to rewrite index file: {e}")
782:         raise
783:
784:
```

```
785:
786: =====
787: FILE: farfan_core/farfancore/models/micro_question_context.py
788: =====
789:
790: """
791: Immutable micro-question context model and deterministic ordering helper.
792: """
793:
794: from __future__ import annotations
795:
796: from dataclasses import dataclass
797: from typing import Any, Iterable
798:
799:
800: @dataclass(frozen=True, slots=True)
801: class MicroQuestionContext:
802:     """
803:         Canonical identity and routing metadata for a micro question.
804:     """
805:
806:     policy_area_id: str
807:     dimension_id: str
808:     question_global: int
809:     question_id: str = ""
810:     base_slot: str | None = None
811:     cluster_id: str | None = None
812:     contract_version: str | None = None
813:     text: str | None = None
814:     patterns: tuple[dict[str, Any], ...] = ()
815:
816:     def __post_init__(self) -> None:
817:         if not self.policy_area_id:
818:             raise ValueError("policy_area_id is required for MicroQuestionContext")
819:         if not self.dimension_id:
820:             raise ValueError("dimension_id is required for MicroQuestionContext")
821:         if self.question_global <= 0:
822:             raise ValueError("question_global must be a positive integer")
823:         normalized_patterns = tuple(self.patterns)
824:         object.__setattr__(self, "patterns", normalized_patterns)
825:         for pattern in normalized_patterns:
826:             pa_in_pattern = pattern.get("policy_area_id")
827:             if pa_in_pattern is None:
828:                 raise ValueError("patterns entries must include policy_area_id")
829:             if pa_in_pattern != self.policy_area_id:
830:                 raise ValueError(
831:                     f"Pattern policy_area_id {pa_in_pattern!r} "
832:                     f"does not match context policy_area_id {self.policy_area_id!r}"
833:                 )
834:
835:
836:     def sort_micro_question_contexts(
837:         questions: Iterable[MicroQuestionContext],
838:     ) -> list[MicroQuestionContext]:
839:         """
840:             Return micro-question contexts deterministically sorted by policy area then global id.
```

```
841: """
842:     return sorted(questions, key=lambda q: (q.policy_area_id, q.question_global))
843:
844:
845: __all__ = ["MicroQuestionContext", "sort_micro_question_contexts"]
846:
847:
848:
849: =====
850: FILE: generate_canonical_inventory.py
851: =====
852:
853: #!/usr/bin/env python3
854: """
855: Generate Canonical Method Inventory with Three Outputs:
856: 1. canonical_method_inventory.json - Full inventory with signatures
857: 2. method_statistics.json - Statistics by role, module, executors
858: 3. excluded_methods.json - Methods flagged as 'never calibrate'
859: """
860:
861: import ast
862: import json
863: from collections import defaultdict
864: from datetime import datetime, timezone
865: from pathlib import Path
866:
867: TRIVIAL_FORMATTERS = {
868:     "__str__",
869:     "__repr__",
870:     "__init__",
871:     "__del__",
872:     "__format__",
873:     "to_string",
874:     "to_json",
875:     "to_dict",
876:     "from_dict",
877:     "__eq__",
878:     "__ne__",
879:     "__hash__",
880:     "__lt__",
881:     "__le__",
882:     "__gt__",
883:     "__ge__",
884: }
885:
886: ROLE_PATTERNS = {
887:     "executor": ["execute", "run_executor", "perform", "apply"],
888:     "orchestrator": ["orchestrate", "coordinate", "run", "execute_suite", "build"],
889:     "analyzer": ["analyze", "infer", "calculate", "compute", "assess"],
890:     "processor": ["process", "transform", "clean", "normalize", "aggregate"],
891:     "extractor": ["extract", "identify", "detect", "find", "locate"],
892:     "scorer": ["score", "evaluate", "rate", "rank", "measure"],
893:     "ingestor": ["parse", "load", "read", "extract_raw", "ingest"],
894:     "utility": ["format", "helper", "validate", "check", "get", "set"],
895:     "core": ["__init__", "setup", "initialize", "configure"],
896: }
```

```
897:  
898:  
899: class CanonicalMethodScanner(ast.NodeVisitor):  
900:     def __init__(self, module_path: str, file_path: str):  
901:         self.module_path = module_path  
902:         self.file_path = file_path  
903:         self.methods = []  
904:         self.current_class = None  
905:         self.class_stack = []  
906:         self.function_depth = 0  
907:  
908:     def visit_ClassDef(self, node):  
909:         self.class_stack.append(node.name)  
910:         self.current_class = ".".join(self.class_stack)  
911:         self.generic_visit(node)  
912:         self.class_stack.pop()  
913:         self.current_class = ".".join(self.class_stack) if self.class_stack else None  
914:  
915:     def visit_FunctionDef(self, node):  
916:         if self.function_depth == 0:  
917:             self._register_method(node)  
918:         self.function_depth += 1  
919:         self.generic_visit(node)  
920:         self.function_depth -= 1  
921:  
922:     def visit_AsyncFunctionDef(self, node):  
923:         if self.function_depth == 0:  
924:             self._register_method(node)  
925:         self.function_depth += 1  
926:         self.generic_visit(node)  
927:         self.function_depth -= 1  
928:  
929:     def _register_method(self, node):  
930:         method_name = node.name  
931:         if self.current_class:  
932:             canonical_name = f"{self.module_path}.{self.current_class}.{method_name}"  
933:         else:  
934:             canonical_name = f"{self.module_path}.{method_name}"  
935:         self.methods.append((canonical_name, node.lineno, self.current_class, node))  
936:  
937:  
938: def extract_signature(node):  
939:     parameters = []  
940:     args = node.args  
941:  
942:     def get_type_hint(arg):  
943:         if arg.annotation:  
944:             try:  
945:                 return ast.unparse(arg.annotation)  
946:             except:  
947:                 return None  
948:         return None  
949:  
950:     def get_default_repr(default):  
951:         try:  
952:             return ast.unparse(default)
```

```
953:         except:
954:             return "..."
955:
956:     all_args = args.args + args.posonlyargs + args.kwonlyargs
957:     defaults = [None] * (len(all_args) - len(args.defaults)) + list(args.defaults)
958:
959:     param_kinds = []
960:     for arg in args.posonlyargs:
961:         param_kinds.append((arg, "POSITIONAL_ONLY"))
962:     for arg in args.args:
963:         param_kinds.append((arg, "POSITIONAL_OR_KEYWORD"))
964:     if args.vararg:
965:         param_kinds.append((args.vararg, "VAR_POSITIONAL"))
966:     for arg in args.kwonlyargs:
967:         param_kinds.append((arg, "KEYWORD_ONLY"))
968:     if args_kwarg:
969:         param_kinds.append((args_kwarg, "VAR_KEYWORD"))
970:
971:     kw_defaults = args_kw_defaults if args_kw_defaults else []
972:
973:     for idx, (arg, kind) in enumerate(param_kinds):
974:         if kind in ("POSITIONAL_ONLY", "POSITIONAL_OR_KEYWORD"):
975:             default_val = defaults[idx] if idx < len(defaults) else None
976:         elif kind == "KEYWORD_ONLY":
977:             try:
978:                 kw_idx = args.kwonlyargs.index(arg)
979:                 default_val = kw_defaults[kw_idx] if kw_idx < len(kw_defaults) else None
980:             except:
981:                 default_val = None
982:         else:
983:             default_val = None
984:
985:         param_info = {
986:             "name": arg.arg,
987:             "kind": kind,
988:             "type_hint": get_type_hint(arg),
989:             "has_default": default_val is not None,
990:             "default_repr": get_default_repr(default_val) if default_val else None,
991:         }
992:         parameters.append(param_info)
993:
994:     return {"parameters": parameters}
995:
996:
997: def classify_role(method_name: str, class_name: str):
998:     method_lower = method_name.lower()
999:     class_lower = (class_name or "").lower()
1000:
1001:    for role, patterns in ROLE_PATTERNS.items():
1002:        for pattern in patterns:
1003:            if pattern.lower() in method_lower:
1004:                return role
1005:
1006:    if "executor" in class_lower:
1007:        return "executor"
1008:    elif "orchestrator" in class_lower:
```

```
1009:         return "orchestrator"
1010:     elif "analyzer" in class_lower or "analyser" in class_lower:
1011:         return "analyzer"
1012:     elif "processor" in class_lower:
1013:         return "processor"
1014:     elif "extractor" in class_lower:
1015:         return "extractor"
1016:     elif "scorer" in class_lower or "scoring" in class_lower:
1017:         return "scorer"
1018:
1019:     if method_name.startswith("_"):
1020:         return "utility"
1021:
1022:     return "core"
1023:
1024:
1025: def is_executor_method(method_name: str, class_name):
1026:     if not class_name:
1027:         return False
1028:     class_lower = class_name.lower()
1029:     return "executor" in class_lower or method_name in (
1030:         "execute",
1031:         "run_executor",
1032:         "perform",
1033:     )
1034:
1035:
1036: def scan_file(file_path, base_path):
1037:     try:
1038:         with open(file_path, encoding="utf-8") as f:
1039:             source = f.read()
1040:             tree = ast.parse(source, filename=str(file_path))
1041:     except:
1042:         return []
1043:
1044:     relative_path = file_path.relative_to(base_path)
1045:     module_parts = list(relative_path.parts[:-1]) + [relative_path.stem]
1046:     module_path = ".".join(module_parts)
1047:
1048:     scanner = CanonicalMethodScanner(module_path, str(file_path.relative_to(base_path)))
1049:     scanner.visit(tree)
1050:
1051:     methods = []
1052:     for canonical_name, line_number, class_name, node in scanner.methods:
1053:         signature = extract_signature(node)
1054:         role = classify_role(node.name, class_name)
1055:         is_exec = is_executor_method(node.name, class_name)
1056:
1057:         methods.append(
1058:             {
1059:                 "canonical_name": canonical_name,
1060:                 "file_path": str(file_path.relative_to(base_path)),
1061:                 "line_number": line_number,
1062:                 "class_name": class_name,
1063:                 "role": role,
1064:                 "is_executor": is_exec,
```

```
1065:             "signature": signature,
1066:         }
1067:     )
1068:
1069:     return methods
1070:
1071:
1072: def scan_directory(directory):
1073:     all_methods = []
1074:     python_files = sorted(directory.rglob("*.py"))
1075:
1076:     for file_path in python_files:
1077:         methods = scan_file(file_path, directory)
1078:         all_methods.extend(methods)
1079:
1080:     return all_methods
1081:
1082:
1083: def generate_excluded_methods(methods):
1084:     excluded = {"excluded_methods": [], "exclusion_reason": "never calibrate"}
1085:
1086:     for method in methods:
1087:         method_name = method["canonical_name"].split(".")[-1]
1088:         if method_name in TRIVIAL_FORMATTERS:
1089:             excluded["excluded_methods"].append(
1090:                 {
1091:                     "canonical_id": method["canonical_name"],
1092:                     "reason": "trivial_formatter",
1093:                     "method_name": method_name,
1094:                     "file_path": method["file_path"],
1095:                     "line_number": method["line_number"],
1096:                 }
1097:             )
1098:         elif method["role"] == "utility" and any(
1099:             x in method_name.lower() for x in ["_format", "_helper", "_to_", "_from_"]
1100:         ):
1101:             excluded["excluded_methods"].append(
1102:                 {
1103:                     "canonical_id": method["canonical_name"],
1104:                     "reason": "utility_formatter",
1105:                     "method_name": method_name,
1106:                     "file_path": method["file_path"],
1107:                     "line_number": method["line_number"],
1108:                 }
1109:             )
1110:
1111:     excluded["total_excluded"] = len(excluded["excluded_methods"])
1112:     return excluded
1113:
1114:
1115: def generate_statistics(methods):
1116:     stats = {
1117:         "total_methods": len(methods),
1118:         "total_executors": sum(1 for m in methods if m["is_executor"]),
1119:         "by_role": defaultdict(int),
1120:         "by_module": defaultdict(int),
```

```
1121:         "executor_distribution": defaultdict(int),
1122:     }
1123:
1124:     for method in methods:
1125:         stats["by_role"][method["role"]] += 1
1126:         module_name = method["canonical_name"].split(".")[0]
1127:         stats["by_module"][module_name] += 1
1128:         if method["is_executor"]:
1129:             stats["executor_distribution"][module_name] += 1
1130:
1131:     stats["by_role"] = dict(stats["by_role"])
1132:     stats["by_module"] = dict(stats["by_module"])
1133:     stats["executor_distribution"] = dict(stats["executor_distribution"])
1134:
1135:     return stats
1136:
1137:
1138: def main():
1139:     source_dir = Path("src/farfan_pipeline")
1140:     if not source_dir.exists():
1141:         print(f"ERROR: {source_dir} does not exist")
1142:         return
1143:
1144:     print(f"Scanning {source_dir}...")
1145:     methods = scan_directory(source_dir)
1146:     print(f"Found {len(methods)} methods")
1147:
1148:     methods_dict = {m["canonical_name"]: m for m in methods}
1149:
1150:     canonical_inventory = {
1151:         "methods": methods_dict,
1152:         "metadata": {
1153:             "total_methods": len(methods),
1154:             "scan_timestamp": datetime.now(timezone.utc).isoformat(),
1155:             "source_directory": str(source_dir),
1156:         },
1157:     }
1158:
1159:     print("Writing canonical_method_inventory.json...")
1160:     with open("canonical_method_inventory.json", "w", encoding="utf-8") as f:
1161:         json.dump(canonical_inventory, f, indent=2, ensure_ascii=False)
1162:     print("Generated canonical_method_inventory.json")
1163:
1164:     statistics = generate_statistics(methods)
1165:     with open("method_statistics.json", "w", encoding="utf-8") as f:
1166:         json.dump(statistics, f, indent=2, ensure_ascii=False)
1167:     print("Generated method_statistics.json")
1168:
1169:     excluded = generate_excluded_methods(methods)
1170:     with open("excluded_methods.json", "w", encoding="utf-8") as f:
1171:         json.dump(excluded, f, indent=2, ensure_ascii=False)
1172:     print("Generated excluded_methods.json")
1173:
1174:     print("\nSummary:")
1175:     print(f"  Total methods: {len(methods)}")
1176:     print(f"  Executors: {statistics['total_executors']}")
```

```
1177:     print(f"  Excluded: {excluded['total_excluded']}")
1178:     print(f"  Roles: {list(statistics['by_role'].keys())}")
1179:
1180:
1181: if __name__ == "__main__":
1182:     main()
1183:
1184:
1185:
1186: =====
1187: FILE: generate_code_pdfs.py
1188: =====
1189:
1190: #!/usr/bin/env python3
1191: """Generate PDF documentation from all Python files in the repository.
1192:
1193: Splits files into 30 organized PDFs for code audit.
1194: """
1195:
1196: import os
1197: import subprocess
1198: from pathlib import Path
1199: from datetime import datetime
1200: from typing import List
1201:
1202: # Configuration
1203: REPO_ROOT = Path(__file__).parent
1204: OUTPUT_DIR = REPO_ROOT / "code_audit_pdfs"
1205: NUM_PDFS = 30
1206: EXCLUDE_DIRS = {"farfan-env", "__pycache__", ".git", "node_modules", ".venv", "venv"}
1207:
1208: def collect_python_files() -> List[Path]:
1209:     """Collect all Python files in the repository."""
1210:     python_files = []
1211:
1212:     for root, dirs, files in os.walk(REPO_ROOT):
1213:         # Filter out excluded directories
1214:         dirs[:] = [d for d in dirs if d not in EXCLUDE_DIRS]
1215:
1216:         for file in files:
1217:             if file.endswith('.py'):
1218:                 python_files.append(Path(root) / file)
1219:
1220:     return sorted(python_files)
1221:
1222: def organize_files_by_directory(files: List[Path]) -> dict[str, List[Path]]:
1223:     """Organize files by their top-level directory."""
1224:     organized = {}
1225:
1226:     for file in files:
1227:         try:
1228:             relative = file.relative_to(REPO_ROOT)
1229:             top_level = str(relative.parts[0]) if len(relative.parts) > 1 else "root"
1230:         except ValueError:
1231:             top_level = "other"
1232:
```

```
1233:         if top_level not in organized:
1234:             organized[top_level] = []
1235:             organized[top_level].append(file)
1236:
1237:     return organized
1238:
1239: def split_into_batches(organized: dict[str, List[Path]], num_batches: int) -> List[List[Path]]:
1240:     """Split files into approximately equal batches."""
1241:     all_files = []
1242:     for files in organized.values():
1243:         all_files.extend(files)
1244:
1245:     batch_size = len(all_files) // num_batches + 1
1246:     batches = []
1247:
1248:     for i in range(0, len(all_files), batch_size):
1249:         batches.append(all_files[i:i + batch_size])
1250:
1251:     return batches
1252:
1253: def create_markdown_for_batch(batch: List[Path], batch_num: int) -> Path:
1254:     """Create a markdown file for a batch of Python files."""
1255:     md_path = OUTPUT_DIR / f"batch_{batch_num:02d}.md"
1256:
1257:     with open(md_path, 'w', encoding='utf-8') as f:
1258:         f.write(f"# F.A.R.F.A.N Pipeline Code Audit - Batch {batch_num}\n\n")
1259:         f.write(f"**Generated**: {datetime.utcnow().isoformat()}\n\n")
1260:         f.write(f"**Files in this batch**: {len(batch)}\n\n")
1261:         f.write("---\n\n")
1262:
1263:         for file_path in batch:
1264:             try:
1265:                 relative = file_path.relative_to(REPO_ROOT)
1266:             except ValueError:
1267:                 relative = file_path
1268:
1269:             f.write(f"## File: '{relative}'\n\n")
1270:             f.write("```python\n")
1271:
1272:             try:
1273:                 with open(file_path, 'r', encoding='utf-8', errors='ignore') as src:
1274:                     content = src.read()
1275:                     f.write(content)
1276:             except Exception as e:
1277:                 f.write(f"# ERROR READING FILE: {e}\n")
1278:
1279:             f.write("```\n\n")
1280:             f.write("---\n\n")
1281:
1282:     return md_path
1283:
1284: def convert_markdown_to_pdf(md_path: Path) -> Path:
1285:     """Convert markdown to PDF using pandoc."""
1286:     pdf_path = md_path.with_suffix('.pdf')
1287:
1288:     cmd = [
```

```
1289:         'pandoc',
1290:         str(md_path),
1291:         '-o', str(pdf_path),
1292:         '--pdf-engine=xelatex',
1293:         '-V', 'geometry:margin=1in',
1294:         '-V', 'fontsize=9pt',
1295:         '-V', 'mainfont=DejaVu Sans Mono',
1296:         '--highlight-style=tango',
1297:     ]
1298:
1299:     try:
1300:         subprocess.run(cmd, check=True, capture_output=True, text=True)
1301:         print(f"\u234\u223 Generated: {pdf_path.name}")
1302:         return pdf_path
1303:     except subprocess.CalledProcessError as e:
1304:         print(f"\u234\u227 Failed to generate PDF for {md_path.name}")
1305:         print(f"  Error: {e.stderr}")
1306:         return None
1307:     except FileNotFoundError:
1308:         print("\u234\u227 pandoc not found. Please install pandoc:")
1309:         print("  brew install pandoc # macOS")
1310:         print("  or visit: https://pandoc.org/installing.html")
1311:         return None
1312:
1313: def check_dependencies() -> bool:
1314:     """Check if required tools are available."""
1315:     try:
1316:         result = subprocess.run(['pandoc', '--version'], capture_output=True, text=True)
1317:         if result.returncode == 0:
1318:             print(f"\u234\u223 pandoc found: {result.stdout.split()[1]}")
1319:             return True
1320:     except FileNotFoundError:
1321:         pass
1322:
1323:     print("\u234\u227 pandoc not found")
1324:     print("\nInstallation instructions:")
1325:     print("  macOS:  brew install pandoc basictex")
1326:     print("  Ubuntu: sudo apt-get install pandoc texlive-xetex")
1327:     print("  Windows: choco install pandoc miktex")
1328:     return False
1329:
1330: def main():
1331:     print("=" * 80)
1332:     print("F.A.R.F.A.N Code Audit PDF Generator")
1333:     print("=" * 80)
1334:     print()
1335:
1336:     # Check dependencies
1337:     if not check_dependencies():
1338:         return 1
1339:
1340:     # Create output directory
1341:     OUTPUT_DIR.mkdir(exist_ok=True)
1342:     print(f"\u234\u223 Output directory: {OUTPUT_DIR}")
1343:     print()
1344:
```

```
1345:     # Collect files
1346:     print("Collecting Python files...")
1347:     all_files = collect_python_files()
1348:     print(f"\u2192 Found {len(all_files)} Python files")
1349:     print()
1350:
1351:     # Organize by directory
1352:     print("Organizing files by directory...")
1353:     organized = organize_files_by_directory(all_files)
1354:     for dir_name, files in sorted(organized.items()):
1355:         print(f"  {dir_name}: {len(files)} files")
1356:     print()
1357:
1358:     # Split into batches
1359:     print(f"Splitting into {NUM_PDFS} batches...")
1360:     batches = split_into_batches(organized, NUM_PDFS)
1361:     actual_batches = len(batches)
1362:     print(f"\u2192 Created {actual_batches} batches")
1363:     for i, batch in enumerate(batches, 1):
1364:         print(f"    Batch {i:02d}: {len(batch)} files")
1365:     print()
1366:
1367:     # Generate PDFs
1368:     print("Generating PDFs...")
1369:     print("(This may take several minutes...)")
1370:     print()
1371:
1372:     successful = 0
1373:     failed = 0
1374:
1375:     for i, batch in enumerate(batches, 1):
1376:         print(f"Processing batch {i}/{actual_batches}...", end=' ')
1377:
1378:         # Create markdown
1379:         md_path = create_markdown_for_batch(batch, i)
1380:
1381:         # Convert to PDF
1382:         pdf_path = convert_markdown_to_pdf(md_path)
1383:
1384:         if pdf_path:
1385:             successful += 1
1386:             # Optionally remove markdown after successful conversion
1387:             # md_path.unlink()
1388:         else:
1389:             failed += 1
1390:
1391:     print()
1392:     print("=" * 80)
1393:     print("SUMMARY")
1394:     print("=" * 80)
1395:     print(f"Total files processed: {len(all_files)}")
1396:     print(f"PDFs generated: {successful}")
1397:     print(f"Failed: {failed}")
1398:     print(f"\nOutput directory: {OUTPUT_DIR}")
1399:     print()
1400:
```

```

1401:     if successful > 0:
1402:         print("â\234\223 PDF generation complete!")
1403:         print("\nYou can now review the code in organized PDF batches.")
1404:
1405:     return 0 if failed == 0 else 1
1406:
1407: if __name__ == '__main__':
1408:     exit(main())
1409:
1410:
1411:
1412: =====
1413: FILE: generate_code_pdbs_simple.py
1414: =====
1415:
1416: #!/usr/bin/env python3
1417: """Generate PDF documentation from all Python files using enscript + ps2pdf.
1418:
1419: Splits files into 30 organized PDFs for code audit.
1420: """
1421:
1422: import os
1423: import subprocess
1424: import tempfile
1425: from pathlib import Path
1426: from datetime import datetime
1427: from typing import List
1428:
1429: # Configuration
1430: REPO_ROOT = Path(__file__).parent
1431: OUTPUT_DIR = REPO_ROOT / "code_audit_pdbs"
1432: NUM_PDOS = 30
1433: EXCLUDE_DIRS = {"farfan-env", "__pycache__", ".git", "node_modules", ".venv", "venv"}
1434:
1435: def collect_python_files() -> List[Path]:
1436:     """Collect all Python files in the repository."""
1437:     python_files = []
1438:
1439:     for root, dirs, files in os.walk(REPO_ROOT):
1440:         dirs[:] = [d for d in dirs if d not in EXCLUDE_DIRS]
1441:
1442:         for file in files:
1443:             if file.endswith('.py'):
1444:                 python_files.append(Path(root) / file)
1445:
1446:     return sorted(python_files)
1447:
1448: def split_into_batches(files: List[Path], num_batches: int) -> List[List[Path]]:
1449:     """Split files into approximately equal batches."""
1450:     batch_size = len(files) // num_batches + 1
1451:     batches = []
1452:
1453:     for i in range(0, len(files), batch_size):
1454:         batches.append(files[i:i + batch_size])
1455:
1456:     return batches

```

```
1457:  
1458: def create_combined_file(batch: List[Path], batch_num: int) -> Path:  
1459:     """Create a combined text file for a batch."""  
1460:     temp_file = OUTPUT_DIR / f"batch_{batch_num:02d}_combined.txt"  
1461:  
1462:     with open(temp_file, 'w', encoding='utf-8') as out:  
1463:         out.write("=" * 80 + "\n")  
1464:         out.write(f"F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH {batch_num}\n")  
1465:         out.write("=" * 80 + "\n")  
1466:         out.write(f"Generated: {datetime.utcnow().isoformat()}\n")  
1467:         out.write(f"Files in this batch: {len(batch)}\n")  
1468:         out.write("=" * 80 + "\n\n")  
1469:  
1470:     for file_path in batch:  
1471:         try:  
1472:             relative = file_path.relative_to(REPO_ROOT)  
1473:         except ValueError:  
1474:             relative = file_path  
1475:  
1476:         out.write("\n" + "=" * 80 + "\n")  
1477:         out.write(f"FILE: {relative}\n")  
1478:         out.write("=" * 80 + "\n\n")  
1479:  
1480:     try:  
1481:         with open(file_path, 'r', encoding='utf-8', errors='ignore') as src:  
1482:             out.write(src.read())  
1483:     except Exception as e:  
1484:         out.write(f"# ERROR READING FILE: {e}\n")  
1485:  
1486:     out.write("\n\n")  
1487:  
1488: return temp_file  
1489:  
1490: def convert_to_pdf(txt_path: Path, batch_num: int) -> Path:  
1491:     """Convert text file to PDF using enscript + ps2pdf."""  
1492:     pdf_path = OUTPUT_DIR / f"FARFAN_Code_Batch_{batch_num:02d}.pdf"  
1493:     ps_path = OUTPUT_DIR / f"batch_{batch_num:02d}.ps"  
1494:  
1495:     try:  
1496:         # Convert to PostScript using enscript  
1497:         enscript_cmd = [  
1498:             'enscript',  
1499:             '--font=Courier8',  
1500:             '--line-numbers',  
1501:             '--columns=1',  
1502:             '--fancy-header',  
1503:             '--landscape',  
1504:             '--output', str(ps_path),  
1505:             str(txt_path)  
1506:         ]  
1507:  
1508:         result = subprocess.run(enscript_cmd, capture_output=True, text=True)  
1509:         if result.returncode != 0:  
1510:             print(f"  à\234\227 enscript failed: {result.stderr}")  
1511:             return None  
1512:
```

```
1513:     # Convert PostScript to PDF
1514:     ps2pdf_cmd = ['ps2pdf', str(ps_path), str(pdf_path)]
1515:     result = subprocess.run(ps2pdf_cmd, capture_output=True, text=True)
1516:
1517:     if result.returncode != 0:
1518:         print(f"  \u27e8\u27e9\u27e8\u27e9 227 ps2pdf failed: {result.stderr}")
1519:     return None
1520:
1521:     # Clean up temporary files
1522:     ps_path.unlink()
1523:     txt_path.unlink()
1524:
1525:     return pdf_path
1526:
1527: except Exception as e:
1528:     print(f"  \u27e8\u27e9\u27e8\u27e9 227 Conversion failed: {e}")
1529:     return None
1530:
1531: def main():
1532:     print("=" * 80)
1533:     print("F.A.R.F.A.N CODE AUDIT PDF GENERATOR")
1534:     print("=" * 80)
1535:     print()
1536:
1537:     # Create output directory
1538:     OUTPUT_DIR.mkdir(exist_ok=True)
1539:     print(f"\u27e8\u27e9\u27e8\u27e9 223 Output directory: {OUTPUT_DIR}")
1540:     print()
1541:
1542:     # Collect files
1543:     print("Collecting Python files...")
1544:     all_files = collect_python_files()
1545:     print(f"\u27e8\u27e9\u27e8\u27e9 223 Found {len(all_files)} Python files")
1546:     print()
1547:
1548:     # Split into batches
1549:     print(f"Splitting into {NUM_PDFS} batches...")
1550:     batches = split_into_batches(all_files, NUM_PDFS)
1551:     actual_batches = len(batches)
1552:     print(f"\u27e8\u27e9\u27e8\u27e9 223 Created {actual_batches} batches")
1553:
1554:     files_per_batch = [len(b) for b in batches]
1555:     print(f"  Files per batch: min={min(files_per_batch)}, max={max(files_per_batch)}, avg={sum(files_per_batch)//len(files_per_batch)}")
1556:     print()
1557:
1558:     # Generate PDFs
1559:     print("Generating PDFs...")
1560:     print("(This will take several minutes...)")
1561:     print()
1562:
1563:     successful = 0
1564:     failed = 0
1565:
1566:     for i, batch in enumerate(batches, 1):
1567:         print(f"[{i:02d}/{actual_batches:02d}] Processing {len(batch)} files...", end=' ', flush=True)
1568:
```

```
1569:         # Create combined text file
1570:         txt_path = create_combined_file(batch, i)
1571:
1572:         # Convert to PDF
1573:         pdf_path = convert_to_pdf(txt_path, i)
1574:
1575:         if pdf_path:
1576:             file_size = pdf_path.stat().st_size / (1024 * 1024)
1577:             print(f"\u234\u223 {pdf_path.name} ({file_size:.1f} MB)")
1578:             successful += 1
1579:         else:
1580:             print(f"\u234\u227 Failed")
1581:             failed += 1
1582:
1583:         print()
1584:         print("=" * 80)
1585:         print("SUMMARY")
1586:         print("=" * 80)
1587:         print(f"Total Python files: {len(all_files)}")
1588:         print(f"Batches created: {actual_batches}")
1589:         print(f"PDFs generated: {successful}")
1590:         print(f"Failed: {failed}")
1591:         print(f"\nOutput directory: {OUTPUT_DIR}")
1592:         print()
1593:
1594:         if successful > 0:
1595:             print("\u234\u223 PDF generation complete!")
1596:             print("\nGenerated files:")
1597:             for pdf in sorted(OUTPUT_DIR.glob("FARFAN_Code_Batch_*.pdf")):
1598:                 size_mb = pdf.stat().st_size / (1024 * 1024)
1599:                 print(f" - {pdf.name} ({size_mb:.1f} MB)")
1600:
1601:     return 0 if failed == 0 else 1
1602:
1603: if __name__ == '__main__':
1604:     exit(main())
1605:
1606:
1607:
1608: =====
1609: FILE: import_contract_validator.py
1610: =====
1611:
1612: """Import Contract Validator - Enforces architectural import boundaries.
1613:
1614: This module validates and enforces the 7 architectural import contracts defined in
1615: pyproject.toml using import-linter. It performs:
1616:
1617: 1. Programmatic execution of import-linter contracts
1618: 2. AST-based import analysis for violation-prone modules
1619: 3. Generation of detailed violation reports with remediation strategies
1620: 4. Failure on any contract violation
1621:
1622: Contracts enforced:
1623: 1. core.calibration/core.wiring must not import analysis/processing/api
1624: 2. core.orchestrator must not import analysis
```

```
1625: 3. processing cannot import orchestrator
1626: 4. analysis cannot import orchestrator
1627: 5. infrastructure must not import orchestrator
1628: 6. api only calls orchestrator entry points
1629: 7. utils stay leaf modules
1630: """
1631:
1632: from __future__ import annotations
1633:
1634: import ast
1635: import subprocess
1636: import sys
1637: from dataclasses import dataclass, field
1638: from pathlib import Path
1639:
1640:
1641: @dataclass
1642: class ImportViolation:
1643:     """Represents a single import contract violation."""
1644:     source_module: str
1645:     imported_module: str
1646:     line_number: int
1647:     import_statement: str
1648:     contract_name: str
1649:     severity: str = "ERROR"
1650:
1651:
1652: @dataclass
1653: class ContractViolationReport:
1654:     """Container for all contract violations found."""
1655:     violations: list[ImportViolation] = field(default_factory=list)
1656:     contracts_checked: int = 0
1657:     contracts_passed: int = 0
1658:     contracts_failed: int = 0
1659:
1660:     def addViolation(self, violation: ImportViolation) -> None:
1661:         self.violations.append(violation)
1662:
1663:     def hasViolations(self) -> bool:
1664:         return len(self.violations) > 0
1665:
1666:
1667: class ASTImportAnalyzer:
1668:     """Analyzes Python source files for import statements using AST."""
1669:
1670:     def __init__(self, project_root: Path) -> None:
1671:         self.project_root = project_root
1672:         self.src_root = project_root / "src" / "farfan_pipeline"
1673:
1674:     def analyze_file(self, file_path: Path) -> list[tuple[str, int, str, int]]:
1675:         try:
1676:             with open(file_path, encoding='utf-8') as f:
1677:                 source = f.read()
1678:                 tree = ast.parse(source, filename=str(file_path))
1679:                 imports = []
1680:                 for node in ast.walk(tree):
```

```

1681:         if isinstance(node, ast.Import):
1682:             for alias in node.names:
1683:                 import_stmt = f"import {alias.name}"
1684:                 imports.append((alias.name, node.lineno, import_stmt, 0))
1685:             elif isinstance(node, ast.ImportFrom) and node.module:
1686:                 level = getattr(node, 'level', 0)
1687:                 dots = '.' * level
1688:                 import_stmt = f"from {dots}{node.module} import {' '.join(a.name for a in node.names)}"
1689:                 imports.append((node.module, node.lineno, import_stmt, level))
1690:             return imports
1691:         except Exception as e:
1692:             print(f"Warning: Failed to analyze {file_path}: {e}")
1693:             return []
1694:
1695:     def get_module_name(self, file_path: Path) -> str:
1696:         try:
1697:             rel_path = file_path.relative_to(self.src_root)
1698:             parts = list(rel_path.parts)
1699:             if parts[-1] == '__init__.py':
1700:                 parts = parts[:-1]
1701:             elif parts[-1].endswith('.py'):
1702:                 parts[-1] = parts[-1][:-3]
1703:             return 'farfan_pipeline.' + '.'.join(parts)
1704:         except ValueError:
1705:             return str(file_path)
1706:
1707:     def check_importViolation(self, source_module: str, imported_module: str, forbidden_patterns: list[str], level: int = 0) -> bool:
1708:         resolved_import = self._resolve_relative_import(source_module, imported_module, level)
1709:         return any(resolved_import.startswith(pattern) for pattern in forbidden_patterns)
1710:
1711:     def _resolve_relative_import(self, source_module: str, imported_module: str, level: int = 0) -> str:
1712:         """Resolve relative imports to absolute module names."""
1713:         if level == 0:
1714:             return imported_module
1715:
1716:         source_parts = source_module.split('.')
1717:         if level >= len(source_parts):
1718:             return imported_module
1719:
1720:         base_parts = source_parts[:len(source_parts) - level]
1721:         if imported_module:
1722:             return '.'.join(base_parts + [imported_module])
1723:         return '.'.join(base_parts)
1724:
1725:
1726:     class ImportContractValidator:
1727:         """Main validator that orchestrates contract checking."""
1728:
1729:         CONTRACTS = [
1730:             {"name": "Core (excluding orchestrator) must not import analysis", "source_patterns": ["farfan_pipeline.core.calibration", "farfan_pipeline.core.wiring"], "forbidden_patterns": ["farfan_pipeline.analysis", "farfan_pipeline.processing", "farfan_pipeline.api"]},
1731:             {"name": "Core orchestrator must not import analysis", "source_patterns": ["farfan_pipeline.core.orchestrator"], "forbidden_patterns": ["farfan_pipeline.analysis"]},
1732:             {"name": "Processing layer cannot import orchestrator", "source_patterns": ["farfan_pipeline.processing"], "forbidden_patterns": ["farfan_pipeline.core.orchestrator"]},
1733:             {"name": "Analysis layer cannot import orchestrator", "source_patterns": ["farfan_pipeline.analysis"], "forbidden_patterns": ["farfan_pipeline.core.wiring"]}]

```

```

orchestrator"]},
1734:     {"name": "Analysis depends on core but not infrastructure", "source_patterns": ["farfan_pipeline.analysis"], "forbidden_patterns": ["farfan_pipeline.infrastructure"]},
1735:     {"name": "Infrastructure must not pull orchestrator", "source_patterns": ["farfan_pipeline.infrastructure"], "forbidden_patterns": ["farfan_pipeline.core.orchestrator"]},
1736:     {"name": "API layer only calls orchestrator entry points", "source_patterns": ["farfan_pipeline.api"], "forbidden_patterns": ["farfan_pipeline.processing", "farfan_pipeline.analysis", "farfan_pipeline.utils"]},
1737:     {"name": "Utils stay leaf modules", "source_patterns": ["farfan_pipeline.utils"], "forbidden_patterns": ["farfan_pipeline.core.orchestrator", "farfan_pipeline.processing", "farfan_pipeline.analysis"]},
1738:   ]
1739:
1740: VIOLATION_PRONE_MODULES = ["src/farfan_pipeline/core/orchestrator", "src/farfan_pipeline/processing/aggregation.py", "src/farfan_pipeline/utils", "src/farfan_pipeline/analysis", "src/farfan_pipeline/api"]
1741:
1742: def __init__(self, project_root: Path | None = None) -> None:
1743:     self.project_root = project_root or Path.cwd()
1744:     self.analyzer = ASTImportAnalyzer(self.project_root)
1745:     self.report = ContractViolationReport()
1746:
1747: def run_import_linter(self) -> tuple[bool, str]:
1748:     try:
1749:         result = subprocess.run(["lint-imports", "--config", "pyproject.toml"], cwd=self.project_root, capture_output=True, text=True, timeout=60, check=False)
1750:         return result.returncode == 0, result.stdout + result.stderr
1751:     except subprocess.TimeoutExpired:
1752:         return False, "import-linter execution timed out"
1753:     except Exception as e:
1754:         return False, f"Failed to run import-linter: {e}"
1755:
1756: def analyze_violation_prone_modules(self) -> None:
1757:     for module_path in self.VIOLATION_PRONE_MODULES:
1758:         full_path = self.project_root / module_path
1759:         if full_path.is_dir():
1760:             for py_file in full_path.rglob("*.py"):
1761:                 self._analyze_single_file(py_file)
1762:         elif full_path.exists():
1763:             self._analyze_single_file(full_path)
1764:
1765: def _analyze_single_file(self, file_path: Path) -> None:
1766:     source_module = self.analyzer.get_module_name(file_path)
1767:     imports = self.analyzer.analyze_file(file_path)
1768:     for imported_module, line_no, import_stmt, level in imports:
1769:         for contract in self.CONTRACTS:
1770:             is_source_match = any(source_module.startswith(pattern) for pattern in contract["source_patterns"])
1771:             if not is_source_match:
1772:                 continue
1773:             is_violation = self.analyzer.check_importViolation(source_module, imported_module, contract["forbidden_patterns"], level)
1774:             if is_violation:
1775:                 violation = ImportViolation(source_module=source_module, imported_module=imported_module, line_number=line_no, import_statement=import_stmt, contract_name=contract["name"])
1776:                 self.report.add_violation(violation)
1777:                 print(f"\u2628\ufe0f Violation: {source_module}:{line_no} imports {imported_module}")
1778:
1779: def _generate_remediation_strategy(self, violation: ImportViolation) -> str:
1780:     strategies = []
1781:     if "orchestrator" in violation.imported_module and "analysis" in violation.source_module:

```

```

1782:     strategies.append("- **Dependency Inversion**: Analysis should not depend on orchestrator. Move shared interfaces to core.contracts.")
1783:     strategies.append("- **Signal-Based Decoupling**: Use the flux signal system for communication instead of direct imports.")
1784:     if "analysis" in violation.imported_module and "orchestrator" in violation.source_module:
1785:         strategies.append("- **Interface Extraction**: Extract analysis interfaces to core layer, have orchestrator depend on interfaces.")
1786:         strategies.append("- **Factory Pattern**: Use a factory in core to instantiate analysis components without direct import.")
1787:         strategies.append("- **Registry Pattern**: Register analysis handlers via a registry mechanism instead of direct coupling.")
1788:     if "processing" in violation.imported_module and "orchestrator" in violation.source_module:
1789:         strategies.append("- **Move to Core**: If orchestrator needs processing functionality, move it to core.processing_contracts.")
1790:         strategies.append("- **Data Flow Inversion**: Pass preprocessed data from orchestrator to processing, not the other way around.")
1791:     if "orchestrator" in violation.imported_module and "processing" in violation.source_module:
1792:         strategies.append("- **Remove Circular Dependency**: Processing should receive data from orchestrator, not import it.")
1793:         strategies.append("- **Use Data Contracts**: Define shared data structures in core.contracts that both can import.")
1794:     if "utils" in violation.source_module:
1795:         strategies.append("- **Keep Utils Lean**: Utils should be leaf modules with no domain dependencies.")
1796:         strategies.append("- **Extract to Core**: If utils needs domain logic, extract it to an appropriate core module.")
1797:     if "api" in violation.source_module:
1798:         strategies.append("- **API Layer Isolation**: API should only call orchestrator entry points, not bypass to lower layers.")
1799:         strategies.append("- **Facade Pattern**: Create orchestrator facades that expose only necessary API operations.")
1800:     if not strategies:
1801:         strategies.append("- **General**: Refactor to respect layer boundaries. Consider dependency injection or interface extraction.")
1802:     return "\n".join(strategies)
1803:
1804: def generate_markdown_report(self) -> str:
1805:     lines = ["# Layer Violation Report", "", "## Summary", "", f"- **Total Contracts Checked**: {len(self.CONTRACTS)}", f"- **Violations Found**: {len(self.report.violations)}", f"- **Status**: {'FAILED' if self.report.has_violations() else 'PASSED'}"]
1806:     if not self.report.has_violations():
1807:         lines.extend(["## All architectural contracts are satisfied!", "", "No import violations detected. The codebase maintains proper layer separation."])
1808:     return "\n".join(lines)
1809: lines.extend(["## Violations Detected", ""])
1810: violations_by_contract: dict[str, list[ImportViolation]] = {}
1811: for violation in self.report.violations:
1812:     if violation.contract_name not in violations_by_contract:
1813:         violations_by_contract[violation.contract_name] = []
1814:         violations_by_contract[violation.contract_name].append(violation)
1815: for contract_name, violations in violations_by_contract.items():
1816:     lines.extend([f"## Contract: {contract_name}", "", f"**Violations**: {len(violations)}"])
1817:     for violation in violations:
1818:         lines.extend([f"### Violation in '{violation.source_module}', ", f"- **Line**: {violation.line_number}", f"- **Import Statement**: '{violation.import_statement}'", f"- **Imported Module**: '{violation.imported_module}'", f"- **Severity**: {violation.severity}", "", "**Remediation Strategy**: ", "", self._generate_remediation_strategy(violation), "", "---"])
1819:         lines.extend(["## Architectural Guidelines", "", "### API Layer", "  calls entry points only", "Core Orchestrator", "  Core (calibration, wiring)", "  Core API", "  Core Utilities", "  Core Analysis", "  Core Orchestrator API", "  Core Orchestrator Utilities", "  Core Orchestrator Analysis", "  Core API Utilities", "  Core API Analysis", "  Core API Orchestrator", "  Core API Utilities Analysis", "  Core API Orchestrator Utilities", "  Core API Orchestrator Analysis"])
1820:     return "\n".join(lines)
1821:
1822: def validate(self) -> int:
1823:     print("=" * 80)
1824:     print("Import Contract Validation")
1825:     print("=" * 80)
1826:     print(f"\nProject Root: {self.project_root}")
1827:     print(f"Contracts to Check: {len(self.CONTRACTS)}\n")
1828:     print("Step 1: Running import-linter...")
1829:     linter_success, linter_output = self.run_import_linter()

```

```
1830:     print(linter_output)
1831:     if not linter_success:
1832:         print("\u232f \u2171 import-linter detected violations\n")
1833:     else:
1834:         print("\u234f\u205c import-linter passed\n")
1835:     print("Step 2: Analyzing violation-prone modules with AST...")
1836:     self.analyzeViolationProneModules()
1837:     print(f"AST Analysis Complete: {len(self.report.violations)} violations found\n")
1838:     print("Step 3: Generating report...")
1839:     report_content = self.generateMarkdownReport()
1840:     report_path = self.project_root / "LAYER_VIOLATION_REPORT.md"
1841:     with open(report_path, 'w', encoding='utf-8') as f:
1842:         f.write(report_content)
1843:     print(f"Report generated: {report_path}\n")
1844:     print("=" * 80)
1845:     if self.report.hasViolations() or not linter_success:
1846:         print("\u235f\u2144 VALIDATION FAILED - Contract violations detected")
1847:         print("=" * 80)
1848:         print(f"\nSee {report_path} for detailed remediation strategies.\n")
1849:     return 1
1850: else:
1851:     print("\u234f\u205c VALIDATION PASSED - All contracts satisfied")
1852:     print("=" * 80)
1853: return 0
1854:
1855:
1856: def main() -> int:
1857:     validator = ImportContractValidator()
1858:     return validator.validate()
1859:
1860:
1861: if __name__ == "__main__":
1862:     sys.exit(main())
1863:
1864:
1865:
1866: =====
1867: FILE: refactor_imports.py
1868: =====
1869:
1870: #!/usr/bin/env python3
1871: """
1872: Refactorizaci\u00f3n autom\u00e1tica de imports relativos a absolutos.
1873: Transforma todos los imports de farfan_core a farfan_pipeline.
1874: """
1875: import ast
1876: import os
1877: import re
1878: from pathlib import Path
1879: from typing import List, Tuple
1880:
1881: # Configuraci\u00f3n
1882: OLD_PACKAGE = "farfan_core"
1883: NEW_PACKAGE = "farfan_pipeline"
1884: SRC_DIR = Path("src/farfan_pipeline")
1885:
```

```
1886: def fix_relative_imports(content: str, module_path: Path) -> str:
1887:     """
1888:         Convierte imports relativos a absolutos.
1889:     """
1890:     # Calculate module name from file path
1891:     rel_path = module_path.relative_to(SRC_DIR)
1892:     parts = list(rel_path.parts[:-1])  # Remove filename
1893:     current_module = ".".join(parts) if parts else ""
1894:
1895:     lines = content.split('\n')
1896:     new_lines = []
1897:
1898:     for line in lines:
1899:         stripped = line.strip()
1900:
1901:         # Match: from . import X or from .. import X
1902:         match = re.match(r'^from (\.\+)(\w+)? import (.+)\$', stripped)
1903:         if match:
1904:             dots = match.group(1)
1905:             module = match.group(2) or ''
1906:             names = match.group(3)
1907:
1908:             level = len(dots)
1909:
1910:             # Calculate absolute module path
1911:             if level == 1:
1912:                 # from .module import X -> from farfan_pipeline.current.module import X
1913:                 if module:
1914:                     if current_module:
1915:                         abs_module = f"{NEW_PACKAGE}.{current_module}.{module}"
1916:                     else:
1917:                         abs_module = f"{NEW_PACKAGE}.{module}"
1918:                 else:
1919:                     # from . import X (same package)
1920:                     if current_module:
1921:                         abs_module = f"{NEW_PACKAGE}.{current_module}"
1922:                     else:
1923:                         abs_module = NEW_PACKAGE
1924:
1925:             else:
1926:                 # Multiple levels up
1927:                 if current_module:
1928:                     parts_list = current_module.split('.')
1929:                     up_levels = level - 1
1930:                     if up_levels >= len(parts_list):
1931:                         parent_module = NEW_PACKAGE
1932:                     else:
1933:                         parent_module = f"{NEW_PACKAGE}{'.'.join(parts_list[:-up_levels])}"
1934:
1935:                 parent_module = NEW_PACKAGE
1936:
1937:             if module:
1938:                 abs_module = f"{parent_module}.{module}"
1939:             else:
1940:                 abs_module = parent_module
1941:
1942:             # Replace line
```

```
1942:         indent = line[:len(line) - len(line.lstrip())]
1943:         new_line = f'{indent}from {abs_module} import {names}'
1944:         new_lines.append(new_line)
1945:     else:
1946:         new_lines.append(line)
1947:
1948:     return '\n'.join(new_lines)
1949:
1950: def fix_old_package_imports(content: str) -> str:
1951:     """
1952:     Reemplaza referencias al paquete antiguo con el nuevo.
1953:     """
1954:     # Replace farfan_core.X with farfan_pipeline.X
1955:     content = re.sub(
1956:         r'\bfafan_core\.',
1957:         f'{NEW_PACKAGE}\.',
1958:         content
1959:     )
1960:
1961:     # Replace "farfan_core" string references
1962:     content = re.sub(
1963:         r'["']farfan_core["']',
1964:         f'"{NEW_PACKAGE}"',
1965:         content
1966:     )
1967:
1968:     return content
1969:
1970: def process_file(filepath: Path) -> Tuple[bool, str]:
1971:     """Process a single Python file."""
1972:     try:
1973:         with open(filepath, 'r', encoding='utf-8') as f:
1974:             original = f.read()
1975:
1976:             # Step 1: Fix relative imports
1977:             modified = fix_relative_imports(original, filepath)
1978:
1979:             # Step 2: Fix old package name
1980:             modified = fix_old_package_imports(modified)
1981:
1982:             if modified != original:
1983:                 with open(filepath, 'w', encoding='utf-8') as f:
1984:                     f.write(modified)
1985:                     return True, "modified"
1986:             else:
1987:                 return False, "unchanged"
1988:
1989:     except Exception as e:
1990:         return False, f"error: {e}"
1991:
1992: def main():
1993:     """Main refactoring process."""
1994:     print(f"Refactorizando imports en {SRC_DIR}")
1995:     print("=" * 80)
1996:
1997:     files_processed = 0
```

```
1998:     files_modified = 0
1999:     files_error = 0
2000:
2001:     for root, dirs, files in os.walk(SRC_DIR):
2002:         # Skip __pycache__
2003:         dirs[:] = [d for d in dirs if d != '__pycache__']
2004:
2005:         for filename in files:
2006:             if filename.endswith('.py'):
2007:                 filepath = Path(root) / filename
2008:                 files_processed += 1
2009:
2010:                 modified, status = process_file(filepath)
2011:
2012:                 if modified:
2013:                     files_modified += 1
2014:                     print(f"\u234\u223 {filepath.relative_to(SRC_DIR)}")
2015:                 elif "error" in status:
2016:                     files_error += 1
2017:                     print(f"\u234\u227 {filepath.relative_to(SRC_DIR)}: {status}")
2018:
2019:     print("=" * 80)
2020:     print(f"Archivos procesados: {files_processed}")
2021:     print(f"Archivos modificados: {files_modified}")
2022:     print(f"Archivos con errores: {files_error}")
2023:     print()
2024:
2025:     if files_modified > 0:
2026:         print("\u234\u223 Refactorizaci\u00f3n completada exitosamente")
2027:         return 0
2028:     else:
2029:         print("\u232 No se realizaron cambios")
2030:         return 1
2031:
2032: if __name__ == "__main__":
2033:     exit(main())
2034:
2035:
2036:
2037: =====
2038: FILE: scan_methods_inventory.py
2039: =====
2040:
2041: #!/usr/bin/env python3
2042: """AST Method Scanner with Canonical Identifier Enforcement
2043:
2044: Traverses src/farfan_pipeline/ recursively, extracts ALL methods as module.Class.method,
2045: classifies by role, applies epistemological rubric, and generates methods_inventory_raw.json.
2046:
2047: FAILURE CONDITION: Aborts with 'insufficient coverage' if <200 entries or if any
2048: pipeline method definition cannot be located.
2049: """
2050:
2051: import ast
2052: import json
2053: import re
```

```
2054: import sys
2055: from dataclasses import asdict, dataclass
2056: from pathlib import Path
2057:
2058: # Extended LAYER_REQUIREMENTS table
2059: LAYER_REQUIREMENTS = {
2060:     "ingest": {
2061:         "description": "Data ingestion and document parsing",
2062:         "typical_patterns": ["parse", "load", "read", "extract_raw", "ingest"],
2063:     },
2064:     "processor": {
2065:         "description": "Data transformation and processing",
2066:         "typical_patterns": ["process", "transform", "clean", "normalize", "aggregate"],
2067:     },
2068:     "analyzer": {
2069:         "description": "Analysis and inference operations",
2070:         "typical_patterns": ["analyze", "infer", "calculate", "compute", "assess"],
2071:     },
2072:     "extractor": {
2073:         "description": "Feature and information extraction",
2074:         "typical_patterns": ["extract", "identify", "detect", "find", "locate"],
2075:     },
2076:     "score": {
2077:         "description": "Scoring and evaluation methods",
2078:         "typical_patterns": ["score", "evaluate", "rate", "rank", "measure"],
2079:     },
2080:     "utility": {
2081:         "description": "Helper and utility functions",
2082:         "typical_patterns": [
2083:             "_format",
2084:             "_helper",
2085:             "_validate",
2086:             "_check",
2087:             "_get",
2088:             "_set",
2089:         ],
2090:     },
2091:     "orchestrator": {
2092:         "description": "Workflow orchestration and coordination",
2093:         "typical_patterns": [
2094:             "orchestrate",
2095:             "coordinate",
2096:             "run",
2097:             "execute_suite",
2098:             "build",
2099:         ],
2100:     },
2101:     "core": {
2102:         "description": "Core framework methods",
2103:         "typical_patterns": ["__init__", "setup", "initialize", "configure"],
2104:     },
2105:     "executor": {
2106:         "description": "Executor pattern implementations",
2107:         "typical_patterns": ["execute", "run_executor", "perform", "apply"],
2108:     },
2109: }
```

```
2110:  
2111:  
2112: @dataclass  
2113: class MethodMetadata:  
2114:     """Metadata for a single method"""  
2115:  
2116:     canonical_identifier: str  
2117:     module_path: str  
2118:     class_name: str | None  
2119:     method_name: str  
2120:     role: str  
2121:     requiere_calibracion: bool  
2122:     requiere_parametrizacion: bool  
2123:     is_async: bool  
2124:     is_property: bool  
2125:     is_classmethod: bool  
2126:     is_staticmethod: bool  
2127:     line_number: int  
2128:     source_file: str  
2129:     epistemology_tags: list[str]  
2130:     is_executor: bool = False  
2131:  
2132:  
2133: class MethodScanner(ast.NodeVisitor):  
2134:     """AST visitor to extract all methods from Python source"""  
2135:  
2136:     def __init__(self, module_path: str, source_file: str) -> None:  
2137:         self.module_path = module_path  
2138:         self.source_file = source_file  
2139:         self.methods: list[MethodMetadata] = []  
2140:         self.current_class: str | None = None  
2141:         self.class_stack: list[str] = []  
2142:         self.function_depth: int = 0  
2143:  
2144:     def visit_ClassDef(self, node: ast.ClassDef) -> None:  
2145:         self.class_stack.append(node.name)  
2146:         self.current_class = ".".join(self.class_stack)  
2147:         self.generic_visit(node)  
2148:         self.class_stack.pop()  
2149:         self.current_class = ".".join(self.class_stack) if self.class_stack else None  
2150:  
2151:     def visit_FunctionDef(self, node: ast.FunctionDef) -> None:  
2152:         if self.function_depth == 0:  
2153:             self._process_function(node, is_async=False)  
2154:         self.function_depth += 1  
2155:         self.generic_visit(node)  
2156:         self.function_depth -= 1  
2157:  
2158:     def visit_AsyncFunctionDef(self, node: ast.AsyncFunctionDef) -> None:  
2159:         if self.function_depth == 0:  
2160:             self._process_function(node, is_async=True)  
2161:         self.function_depth += 1  
2162:         self.generic_visit(node)  
2163:         self.function_depth -= 1  
2164:  
2165:     def _process_function(  
2166:
```

```
2166:         self, node: ast.FunctionDef | ast.AsyncFunctionDef, is_async: bool
2167:     ) -> None:
2168:         method_name = node.name
2169:
2170:         if self.current_class:
2171:             canonical_id = f"{self.module_path}.{self.current_class}.{method_name}"
2172:             class_name = self.current_class
2173:         else:
2174:             canonical_id = f"{self.module_path}.{method_name}"
2175:             class_name = None
2176:
2177:         is_property = any(
2178:             self._is_decorator(d, "property") for d in node.decorator_list
2179:         )
2180:         is_classmethod = any(
2181:             self._is_decorator(d, "classmethod") for d in node.decorator_list
2182:         )
2183:         is_staticmethod = any(
2184:             self._is_decorator(d, "staticmethod") for d in node.decorator_list
2185:         )
2186:
2187:         role, is_executor = self._classify_role(
2188:             method_name, class_name, canonical_id, self.source_file
2189:         )
2190:         requires_calibration, requires_parametrization, epi_tags = (
2191:             self._apply_epistemological_rubric(method_name, class_name, role)
2192:         )
2193:
2194:         metadata = MethodMetadata(
2195:             canonical_identifier=canonical_id,
2196:             module_path=self.module_path,
2197:             class_name=class_name,
2198:             method_name=method_name,
2199:             role=role,
2200:             requiere_calibracion=requires_calibration,
2201:             requiere_parametrizacion=requires_parametrization,
2202:             is_async=is_async,
2203:             is_property=is_property,
2204:             is_classmethod=is_classmethod,
2205:             is_staticmethod=is_staticmethod,
2206:             line_number=node.lineno,
2207:             source_file=self.source_file,
2208:             epistemology_tags=epi_tags,
2209:             is_executor=is_executor,
2210:         )
2211:
2212:         self.methods.append(metadata)
2213:
2214:     def _is_decorator(self, decorator: ast.expr, name: str) -> bool:
2215:         if isinstance(decorator, ast.Name):
2216:             return decorator.id == name
2217:         elif isinstance(decorator, ast.Attribute):
2218:             return decorator.attr == name
2219:         return False
2220:
2221:     def _classify_role(
```

```
2222:         self,
2223:         method_name: str,
2224:         class_name: str | None,
2225:         canonical_name: str,
2226:         source_file: str,
2227:     ) -> tuple[str, bool]:
2228:         method_lower = method_name.lower()
2229:         class_lower = class_name.lower() if class_name else ""
2230:
2231:         executor_pattern = re.compile(r"D[1-6]_?Q[1-5]", re.IGNORECASE)
2232:         is_executor = bool(
2233:             executor_pattern.search(canonical_name)
2234:             or (class_name and executor_pattern.search(class_name))
2235:         )
2236:
2237:         if any(
2238:             kw in method_lower
2239:             for kw in ["parse", "load", "ingest", "read_doc", "extract_raw"]
2240:         ):
2241:             return "ingest", is_executor
2242:
2243:         if any(
2244:             kw in method_lower
2245:             for kw in ["process", "transform", "clean", "normalize", "aggregate"]
2246:         ):
2247:             return "processor", is_executor
2248:
2249:         if any(
2250:             kw in method_lower
2251:             for kw in ["analyze", "infer", "calculate", "compute", "assess"]
2252:         ):
2253:             return "analyzer", is_executor
2254:
2255:         if any(
2256:             kw in method_lower
2257:             for kw in ["extract", "identify", "detect", "find", "locate"]
2258:         ):
2259:             return "extractor", is_executor
2260:
2261:         if any(
2262:             kw in method_lower
2263:             for kw in ["score", "grading", "evaluate", "rate", "rank", "measure"]
2264:         ):
2265:             return "score", is_executor
2266:
2267:         if any(
2268:             kw in method_lower
2269:             for kw in [
2270:                 "orchestrate",
2271:                 "pipeline",
2272:                 "run_all",
2273:                 "coordinate",
2274:                 "execute_suite",
2275:             ]
2276:         ):
2277:             return "orchestrator", is_executor
```

```
2278:
2279:     if (is_executor or "execute" in method_lower) and (
2280:         executor_pattern.search(canonical_name)
2281:         or (class_name and executor_pattern.search(class_name)))
2282:     ):
2283:         return "executor", True
2284:
2285:     if "/core/" in source_file.replace("\\\\", "/") or "core" in method_lower:
2286:         return "core", is_executor
2287:
2288:     if "executor" in class_lower:
2289:         return "executor", True
2290:     elif "aggregator" in class_lower or "processor" in class_lower:
2291:         return "processor", is_executor
2292:     elif "analyzer" in class_lower:
2293:         return "analyzer", is_executor
2294:     elif "extractor" in class_lower:
2295:         return "extractor", is_executor
2296:     elif "scorer" in class_lower:
2297:         return "score", is_executor
2298:     elif "orchestrator" in class_lower:
2299:         return "orchestrator", is_executor
2300:
2301:     if method_name.startswith("_"):
2302:         return "utility", is_executor
2303:
2304:     return "utility", is_executor
2305:
2306: def _apply_epistemological_rubric(
2307:     self, method_name: str, class_name: str | None, role: str
2308: ) -> tuple[bool, bool, list[str]]:
2309:     method_lower = method_name.lower()
2310:     class_lower = class_name.lower() if class_name else ""
2311:     epi_tags = []
2312:
2313:     evaluative_keywords = [
2314:         "score",
2315:         "evaluate",
2316:         "assess",
2317:         "rank",
2318:         "rate",
2319:         "judge",
2320:         "validate",
2321:     ]
2322:     is_evaluative = any(kw in method_lower for kw in evaluative_keywords)
2323:
2324:     transformation_keywords = [
2325:         "calculate",
2326:         "compute",
2327:         "infer",
2328:         "estimate",
2329:         "analyze",
2330:         "transform",
2331:         "process",
2332:         "aggregate",
2333:         "bayesian",
```

```
2334:         ]
2335:         is_transformation = any(kw in method_lower for kw in transformation_keywords)
2336:
2337:         statistical_keywords = [
2338:             "probability",
2339:             "likelihood",
2340:             "confidence",
2341:             "threshold",
2342:             "statistical",
2343:         ]
2344:         is_statistical = any(kw in method_lower for kw in statistical_keywords)
2345:
2346:         is_direct_impact = role in [
2347:             "analyzer",
2348:             "processor",
2349:             "score",
2350:             "executor",
2351:         ] and not method_name.startswith("_")
2352:
2353:         if is_evaluative:
2354:             epi_tags.append("evaluative_judgment")
2355:         if is_transformation:
2356:             epi_tags.append("transformation")
2357:         if is_statistical:
2358:             epi_tags.append("statistical")
2359:         if "causal" in method_lower or "causal" in class_lower:
2360:             epi_tags.append("causal")
2361:         if "bayesian" in method_lower or "bayesian" in class_lower:
2362:             epi_tags.append("bayesian")
2363:         if role == "score":
2364:             epi_tags.append("normative")
2365:         if role == "ingest":
2366:             epi_tags.append("structural")
2367:         if role == "extractor":
2368:             epi_tags.append("semantic")
2369:         if (
2370:             "build" in method_lower
2371:             or "create" in method_lower
2372:             or "generate" in method_lower
2373:         ):
2374:             epi_tags.append("constructive")
2375:         if (
2376:             "check" in method_lower
2377:             or "verify" in method_lower
2378:             or "assert" in method_lower
2379:         ):
2380:             epi_tags.append("consistency")
2381:         if (
2382:             "format" in method_lower
2383:             or "render" in method_lower
2384:             or "export" in method_lower
2385:         ):
2386:             epi_tags.append("descriptive")
2387:
2388:         requires_calibration = is_evaluative or (is_statistical and is_direct_impact)
2389:         requires_parametrization = (
```

```
2390:             is_transformation or is_statistical or (role == "analyzer")
2391:         )
2392:
2393:         if role == "utility" and not is_statistical:
2394:             requires_calibration = False
2395:             requires_parametrization = False
2396:
2397:         if method_name in ["__init__", "__repr__", "__str__"]:
2398:             requires_calibration = False
2399:             requires_parametrization = False
2400:
2401:     return requires_calibration, requires_parametrization, epi_tags
2402:
2403:
2404: def scan_python_file(file_path: Path, base_path: Path) -> list[MethodMetadata]:
2405:     try:
2406:         with open(file_path, encoding="utf-8") as f:
2407:             source = f.read()
2408:
2409:         tree = ast.parse(source, filename=str(file_path))
2410:
2411:         relative_path = file_path.relative_to(base_path)
2412:         module_parts = list(relative_path.parts[:-1]) + [relative_path.stem]
2413:         module_path = ".".join(module_parts)
2414:
2415:         scanner = MethodScanner(module_path, str(file_path))
2416:         scanner.visit(tree)
2417:
2418:         return scanner.methods
2419:
2420:     except SyntaxError as e:
2421:         print(f"WARNING: Syntax error in {file_path}: {e}", file=sys.stderr)
2422:         return []
2423:     except Exception as e:
2424:         print(f"WARNING: Error processing {file_path}: {e}", file=sys.stderr)
2425:         return []
2426:
2427:
2428: def scan_directory(directory: Path) -> list[MethodMetadata]:
2429:     all_methods = []
2430:     python_files = sorted(directory.rglob("*.py"))
2431:
2432:     print(f"Scanning {len(python_files)} Python files...")
2433:
2434:     for file_path in python_files:
2435:         methods = scan_python_file(file_path, directory)
2436:         all_methods.extend(methods)
2437:         print(f"  {file_path.name}: {len(methods)} methods")
2438:
2439:     all_methods = deduplicate_canonical_ids(all_methods)
2440:
2441:     return all_methods
2442:
2443:
2444: def deduplicate_canonical_ids(methods: list[MethodMetadata]) -> list[MethodMetadata]:
2445:     """Deduplicate canonical IDs by appending line numbers to duplicates"""

```

```
2446:     id_counts: dict[str, int] = {}
2447:     for method in methods:
2448:         cid = method.canonical_identifier
2449:         id_counts[cid] = id_counts.get(cid, 0) + 1
2450:
2451:     id_counters: dict[str, int] = {}
2452:     deduplicated = []
2453:
2454:     for method in methods:
2455:         cid = method.canonical_identifier
2456:         if id_counts[cid] > 1:
2457:             id_counters[cid] = id_counters.get(cid, 0) + 1
2458:             new_cid = f"{cid}@L{method.line_number}"
2459:             method.canonical_identifier = new_cid
2460:             deduplicated.append(method)
2461:
2462:     return deduplicated
2463:
2464:
2465: def verify_critical_methods(methods: list[MethodMetadata]) -> tuple[bool, list[str]]:
2466:     critical_files = [
2467:         "derek_beach.py",
2468:         "aggregation.py",
2469:         "executors.py",
2470:         "executors_contract.py",
2471:     ]
2472:
2473:     missing_methods = []
2474:     found_count = 0
2475:
2476:     for method in methods:
2477:         if any(cf in method.source_file for cf in critical_files):
2478:             found_count += 1
2479:
2480:     critical_method_patterns = [
2481:         "derek_beach",
2482:         "aggregation",
2483:         "executors",
2484:         "executors_contract",
2485:     ]
2486:
2487:     found_patterns = set()
2488:     for method in methods:
2489:         for pattern in critical_method_patterns:
2490:             if pattern in method.canonical_identifier.lower():
2491:                 found_patterns.add(pattern)
2492:
2493:     for pattern in critical_method_patterns:
2494:         if pattern not in found_patterns:
2495:             missing_methods.append(f"No methods found matching pattern: {pattern}")
2496:
2497:     if found_count == 0:
2498:         missing_methods.append("CRITICAL: No methods found from any critical file")
2499:
2500:     return len(missing_methods) == 0, missing_methods
2501:
```

```
2502:  
2503: def generate_inventory(methods: list[MethodMetadata], output_file: str) -> None:  
2504:     by_role: dict[str, int] = {}  
2505:     by_file: dict[str, int] = {}  
2506:  
2507:     for method in methods:  
2508:         role = method.role  
2509:         by_role[role] = by_role.get(role, 0) + 1  
2510:  
2511:         file_name = Path(method.source_file).name  
2512:         by_file[file_name] = by_file.get(file_name, 0) + 1  
2513:  
2514:     inventory = {  
2515:         "metadata": {  
2516:             "total_methods": len(methods),  
2517:             "scan_timestamp": None,  
2518:             "source_directory": "src/farfan_pipeline",  
2519:         },  
2520:         "layer_requirements": LAYER_REQUIREMENTS,  
2521:         "methods": [asdict(m) for m in methods],  
2522:         "statistics": {  
2523:             "by_role": by_role,  
2524:             "by_file": by_file,  
2525:             "requiring_calibration": sum(1 for m in methods if m.requiere_calibracion),  
2526:             "requiring_parametrization": sum(  
2527:                 1 for m in methods if m.requiere_parametrizacion  
2528:             ),  
2529:             "executor_count": sum(1 for m in methods if m.is_executor),  
2530:         },  
2531:     }  
2532:  
2533:     with open(output_file, "w", encoding="utf-8") as f:  
2534:         json.dump(inventory, f, indent=2, ensure_ascii=False)  
2535:  
2536:     print(f"\nInventory written to {output_file}")  
2537:     print(f"Total methods: {len(methods)}")  
2538:     print(f"By role: {by_role}")  
2539:     print(f"Requiring calibration: {sum(1 for m in methods if m.requiere_calibracion)}")  
2540:     print(  
2541:         f"Requiring parametrization: {sum(1 for m in methods if m.requiere_parametrizacion)}")  
2542:     )  
2543:     print(f"D1Q1-D6Q5 executors detected: {sum(1 for m in methods if m.is_executor)}")  
2544:  
2545:  
2546: def main() -> None:  
2547:     pipeline_dir = Path("src/farfan_pipeline")  
2548:  
2549:     if not pipeline_dir.exists():  
2550:         print(f"ERROR: Directory {pipeline_dir} does not exist", file=sys.stderr)  
2551:         sys.exit(1)  
2552:  
2553:     print(f"Starting AST method scan of {pipeline_dir}...")  
2554:     methods = scan_directory(pipeline_dir)  
2555:  
2556:     print(f"\n{'='*60})  
2557:     print(f"SCAN COMPLETE: Found {len(methods)} methods")
```

```
2558:     print(f'='*60)")
2559:
2560:     if len(methods) < 200:
2561:         print(
2562:             f"\nERROR: Insufficient coverage - found only {len(methods)} methods (minimum: 200)"
2563:             file=sys.stderr,
2564:         )
2565:         sys.exit(1)
2566:
2567:     verification_passed, missing = verify_critical_methods(methods)
2568:
2569:     if not verification_passed:
2570:         print("\nERROR: Critical method verification failed:", file=sys.stderr)
2571:         for msg in missing:
2572:             print(f" - {msg}", file=sys.stderr)
2573:         sys.exit(1)
2574:
2575:     print("\n\u2344\u2344\u2344 Critical method verification passed")
2576:
2577:     output_file = "methods_inventory_raw.json"
2578:     generate_inventory(methods, output_file)
2579:
2580:     print(f'\n{'='*60}")
2581:     print("SUCCESS: Inventory generation complete")
2582:     print(f'{'='*60}")
2583:
2584:
2585: if __name__ == "__main__":
2586:     main()
2587:
2588:
2589:
2590: =====
2591: FILE: scripts/__init__.py
2592: =====
2593:
2594:
2595:
2596:
2597: =====
2598: FILE: scripts/analyze_requirements.py
2599: =====
2600:
2601: #!/usr/bin/env python3
2602: """
2603: Complete requirements compatibility analyzer.
2604: Checks ALL packages against Python 3.10.12 and identifies conflicts.
2605: """
2606: import sys
2607: import json
2608: from packaging import version
2609: from packaging.requirements import Requirement
2610:
2611: def parse_requirements_file(filepath):
2612:     """Parse requirements file and return list of requirements."""
2613:     requirements = []
```

```
2614:     with open(filepath, 'r') as f:
2615:         for line in f:
2616:             line = line.strip()
2617:             # Skip comments and empty lines
2618:             if line and not line.startswith('#') and not line.startswith('-r'):
2619:                 try:
2620:                     req = Requirement(line)
2621:                     requirements.append({
2622:                         'name': req.name,
2623:                         'specifier': str(req.specifier),
2624:                         'raw': line,
2625:                         'file': filepath
2626:                     })
2627:                 except Exception as e:
2628:                     print(f"Warning: Could not parse '{line}': {e}", file=sys.stderr)
2629:             return requirements
2630:
2631: def main():
2632:     files = [
2633:         'requirements.txt',
2634:         'requirements-dev.txt',
2635:         'requirements-optional.txt',
2636:         'requirements-docs.txt'
2637:     ]
2638:
2639:     all_requirements = {}
2640:     conflicts = []
2641:
2642:     # Parse all files
2643:     for filepath in files:
2644:         try:
2645:             reqs = parse_requirements_file(filepath)
2646:             for req in reqs:
2647:                 name = req['name'].lower()
2648:                 if name not in all_requirements:
2649:                     all_requirements[name] = []
2650:                     all_requirements[name].append(req)
2651:             except FileNotFoundError:
2652:                 print(f"Warning: {filepath} not found", file=sys.stderr)
2653:
2654:     # Find conflicts
2655:     print("=" * 80)
2656:     print("REQUIREMENTS COMPATIBILITY ANALYSIS")
2657:     print("=" * 80)
2658:     print(f"\nPython Version: 3.10.12\n")
2659:
2660:     print("DUPLICATE PACKAGES (different versions):")
2661:     print("-" * 80)
2662:     for name, reqs in sorted(all_requirements.items()):
2663:         if len(reqs) > 1:
2664:             # Check if versions differ
2665:             versions = set(req['specifier'] for req in reqs)
2666:             if len(versions) > 1:
2667:                 conflicts.append(name)
2668:                 print(f"\n\t{i}\t{name.upper()}")
2669:                 for req in reqs:
```

```
2670:             print(f"      {req['file']}:{30s} \u206222 {req['specifier']}")  
2671:  
2672:     if not conflicts:  
2673:         print(" \u234\u223 No version conflicts found")  
2674:  
2675: # Known Python 3.10 incompatibilities  
2676: print("\n\nKNOWN PYTHON 3.10 INCOMPATIBILITIES:")  
2677: print("-" * 80)  
2678:  
2679: py310_issues = {  
2680:     'networkx': ('3.5', '3.4.2', 'networkx 3.5 requires Python >=3.11'),  
2681:     'sentence-transformers': ('3.3.1', '3.0.1', 'Newer versions may have better compatibility'),  
2682: }  
2683:  
2684: for name, reqs in sorted(all_requirements.items()):  
2685:     if name in py310_issues:  
2686:         pinned_ver, recommended_ver, reason = py310_issues[name]  
2687:         for req in reqs:  
2688:             if pinned_ver in req['specifier']:  
2689:                 print(f"\u235\u214 {name.upper()}")  
2690:                 print(f" Current: {req['specifier']}")  
2691:                 print(f" Recommended for Python 3.10: =={recommended_ver}")  
2692:                 print(f" Reason: {reason}")  
2693:  
2694:     print("\n\nRECOMMENDED FIXES:")  
2695:     print("==" * 80)  
2696:     print("")  
2697: 1. networkx: Change from ==3.5 to ==3.4.2 (last version supporting Python 3.10)  
2698: 2. Flask: Consolidate to ==3.0.3 (resolve conflict between requirements.txt and requirements-optional.txt)  
2699: 3. flask-socketio: Use ==5.4.1 (from requirements-optional.txt)  
2700: 4. pyjwt: Use ==2.10.1 (from requirements-optional.txt)  
2701: 5. sentence-transformers: Keep ==3.0.1 for stability  
2702:  
2703: All other packages appear compatible with Python 3.10.12  
2704: """)  
2705:  
2706:     return 0 if not conflicts else 1  
2707:  
2708: if __name__ == '__main__':  
2709:     sys.exit(main())  
2710:  
2711:  
2712:  
2713: =====  
2714: FILE: scripts/audit_hardcoded_values.py  
2715: =====  
2716:  
2717: #!/usr/bin/env python3  
2718: import os  
2719: import re  
2720: import ast  
2721: import sys  
2722: from pathlib import Path  
2723:  
2724: # Configuration  
2725: ROOT_DIR = Path(__file__).parent.parent
```

```
2726: SRC_DIR = ROOT_DIR / "src"
2727: VIOLATIONS_REPORT = ROOT_DIR / "violations_audit.md"
2728:
2729: # Patterns to flag
2730: HARDCODED_FLOAT_PATTERN = re.compile(r"=\s*(0\.\d+|1\.)\b")
2731: CALIBRATION_KEYWORDS = ["weight", "score", "threshold", "calibration", "alpha", "beta", "gamma"]
2732: YAML_EXTENSION_PATTERN = re.compile(r"\.ya?ml", re.IGNORECASE)
2733:
2734: class CalibrationVisitor(ast.NodeVisitor):
2735:     def __init__(self, filename):
2736:         self.filename = filename
2737:         self.violations = []
2738:
2739:     def visit_Assign(self, node):
2740:         # Check for hardcoded float assignments to suspicious variables
2741:         for target in node.targets:
2742:             if isinstance(target, ast.Name):
2743:                 if any(kw in target.id.lower() for kw in CALIBRATION_KEYWORDS):
2744:                     if isinstance(node.value, ast.Constant) and isinstance(node.value.value, float):
2745:                         if 0.0 <= node.value.value <= 1.0:
2746:                             self.violations.append(
2747:                                 f"Hardcoded calibration value detected: {target.id} = {node.value.value} (Line {node.lineno})"
2748:                             )
2749:             self.generic_visit(node)
2750:
2751:     def visit_Dict(self, node):
2752:         # Check for dict literals that look like calibration configs
2753:         has_calibration_keys = False
2754:         for key in node.keys:
2755:             if isinstance(key, ast.Constant) and isinstance(key.value, str):
2756:                 if any(kw in key.value.lower() for kw in CALIBRATION_KEYWORDS):
2757:                     has_calibration_keys = True
2758:                     break
2759:
2760:         if has_calibration_keys:
2761:             for value in node.values:
2762:                 if isinstance(value, ast.Constant) and isinstance(value.value, float):
2763:                     if 0.0 <= value.value <= 1.0:
2764:                         self.violations.append(
2765:                             f"Potential hardcoded calibration dict detected at Line {node.lineno}"
2766:                         )
2767:                     break
2768:             self.generic_visit(node)
2769:
2770:     def visit_Call(self, node):
2771:         # Check for yaml loading or usage
2772:         if isinstance(node.func, ast.Attribute):
2773:             if node.func.attr == 'safe_load' or node.func.attr == 'load':
2774:                 # This is a weak check, but catches yaml.safe_load
2775:                 pass
2776:             self.generic_visit(node)
2777:
2778: def scan_file(filepath):
2779:     violations = []
2780:     try:
2781:         with open(filepath, "r", encoding="utf-8") as f:
```

```
2782:         content = f.read()
2783:
2784:     # Regex checks
2785:     if YAML_EXTENSION_PATTERN.search(content):
2786:         violations.append("Reference to YAML file detected (YAML is prohibited)")
2787:
2788:     # AST checks
2789:     try:
2790:         tree = ast.parse(content, filename=filepath)
2791:         visitor = CalibrationVisitor(filepath)
2792:         visitor.visit(tree)
2793:         violations.extend(visitor.violations)
2794:     except SyntaxError:
2795:         violations.append("SyntaxError: Could not parse file for AST analysis")
2796:
2797:     except Exception as e:
2798:         violations.append(f"Error scanning file: {str(e)}")
2799:
2800:     return violations
2801:
2802: def main():
2803:     print(f"Starting audit of {SRC_DIR}...")
2804:     all_violations = {}
2805:
2806:     if not SRC_DIR.exists():
2807:         print(f"Source directory {SRC_DIR} not found!")
2808:         return
2809:
2810:     for filepath in SRC_DIR.rglob("*.py"):
2811:         violations = scan_file(filepath)
2812:         if violations:
2813:             rel_path = filepath.relative_to(ROOT_DIR)
2814:             all_violations[str(rel_path)] = violations
2815:
2816:     # Generate Report
2817:     with open(VIOLATIONS_REPORT, "w", encoding="utf-8") as f:
2818:         f.write("# SIN_CARRETA Compliance Audit Report\n\n")
2819:         f.write("## Violations Detected\n\n")
2820:
2821:         if not all_violations:
2822:             f.write("No violations found. System is compliant.\n")
2823:         else:
2824:             for path, issues in all_violations.items():
2825:                 f.write(f"## {path}\n")
2826:                 for issue in issues:
2827:                     f.write(f"- {issue}\n")
2828:                 f.write("\n")
2829:
2830:     print(f"Audit complete. Report generated at {VIOLATIONS_REPORT}")
2831:     if all_violations:
2832:         sys.exit(1)
2833:     else:
2834:         sys.exit(0)
2835:
2836: if __name__ == "__main__":
2837:     main()
```

12/07/25

01:17:14

/Users/recovered/Applications/F.A.R.F.A.N -MECHANISTIC-PIPELINE/code_audit_pdfs/batch_01_combined.txt

2838:

2839:

52