```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 4
================================================================================
Generated: 2025-12-07T06:17:15.681070
Files in this batch: 17
================================================================================


================================================================================
FILE: scripts/generate_coverage_report.py
================================================================================

#!/usr/bin/env python3
"""
generate_coverage_report.py - Generate an HTML report for questionnaire coverage gaps.

This script takes the audit_manifest.json as input and generates a user-friendly
HTML report that visualizes the contract coverage gaps. The report includes
overall metrics and a breakdown by dimension and policy area.
"""

import json
from pathlib import Path
from typing import Any, Dict, List

PROJECT_ROOT = Path(__file__).parent.parent.resolve()
AUDIT_MANIFEST_PATH = PROJECT_ROOT / "artifacts" / "audit" / "audit_manifest.json"
MONOLITH_PATH = PROJECT_ROOT / "data" / "questionnaire_monolith.json"
OUTPUT_DIR = PROJECT_ROOT / "artifacts" / "audit"
HTML_REPORT_PATH = OUTPUT_DIR / "coverage_report.html"

HTML_TEMPLATE = """
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Questionnaire Coverage Report</title>
    <style>
        body {{ font-family: sans-serif; margin: 2em; }}
        h1, h2 {{ color: #333; }}
        table {{ border-collapse: collapse; width: 100%; margin-bottom: 2em; }}
        th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
        th {{ background-color: #f2f2f2; }}
        .summary {{ background-color: #f9f9f9; padding: 1em; border-radius: 5px; margin-bottom: 2em; }}
        .summary-metric {{ display: inline-block; margin-right: 2em; }}
        .summary-metric h3 {{ margin: 0; color: #555; }}
        .summary-metric p {{ margin: 0; font-size: 2em; font-weight: bold; }}
        .coverage-bar {{ background-color: #e0e0e0; border-radius: 3px; overflow: hidden; height: 20px; }}
        .coverage-fill {{ background-color: #4CAF50; height: 100%; }}
        .gap-list {{ column-count: 3; }}
        .gap-list li {{ margin-bottom: 0.5em; }}
    </style>
</head>
<body>
    <h1>Questionnaire Coverage Report</h1>
```

```
    <div class="summary">
        <h2>Executive Summary</h2>
        <div class="summary-metric">
            <h3>Total Micro-Questions</h3>
            <p>{total_micro_questions}</p>
        </div>
        <div class="summary-metric">
            <h3>Contract Coverage</h3>
            <p>{contract_coverage_percentage:.2f}%</p>
        </div>
        <div class="summary-metric">
            <h3>Questions with Contract</h3>
            <p>{questions_with_contract}</p>
        </div>
        <div class="summary-metric">
            <h3>Questions without Contract</h3>
            <p>{questions_without_contract}</p>
        </div>
    </div>

    <h2>Contract Coverage Details</h2>
    <div class="coverage-bar">
        <div class="coverage-fill" style="width: {contract_coverage_percentage:.2f}%;"></div>
    </div>
    <p>{questions_with_contract} of {total_micro_questions} questions have contracts.</p>

    <h2>Missing Contracts</h2>
    <p>There are {num_missing_unique} unique questions missing contracts across all policy areas.</p>
    <div class="gap-list">
        <ul>
            {missing_contracts_list}
        </ul>
    </div>
</body>
</html>
"""

def generate_report():
    """
    Generates the HTML report from the audit manifest.
    """
    print("Generating HTML coverage report...")

    # Load audit manifest
    if not AUDIT_MANIFEST_PATH.exists():
        print(f"Error: Audit manifest not found at {AUDIT_MANIFEST_PATH}")
        return
    manifest = json.loads(AUDIT_MANIFEST_PATH.read_text(encoding="utf-8"))

    metrics = manifest.get("metrics", {})
    gaps = manifest.get("gaps", {})

    missing_contracts_list_items = "".join(
        f"<li>{qid}</li>" for qid in gaps.get("questions_without_contract_details", [])
    )

```

```
    html_content = HTML_TEMPLATE.format(
        total_micro_questions=metrics.get("total_micro_questions", 0),
        contract_coverage_percentage=metrics.get("contract_coverage_percentage", 0),
        questions_with_contract=metrics.get("questions_with_contract", 0),
        questions_without_contract=metrics.get("questions_without_contract", 0),
        num_missing_unique=len(gaps.get("questions_without_contract_details", [])),
        missing_contracts_list=missing_contracts_list_items,
    )

    # Write HTML report
    OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
    with open(HTML_REPORT_PATH, "w", encoding="utf-8") as f:
        f.write(html_content)

    print(f"HTML report generated successfully at {HTML_REPORT_PATH}")

if __name__ == "__main__":
    generate_report()



================================================================================
FILE: scripts/generate_layer_inventory.py
================================================================================

#!/usr/bin/env python3
"""
Generate canonical layer inventory JSON configuration.

This script generates config/canonic_inventorry_methods_layers.json
with layer assignments and Choquet weights for all executors.
"""

import json
import sys
from pathlib import Path

from farfan_pipeline.core.calibration.layer_assignment import (
    generate_canonical_inventory,
)

repo_root = Path(__file__).parent.parent


def main():
    executors_file = (
        repo_root / "src" / "farfan_pipeline" / "core" / "orchestrator" / "executors.py"
    )
    output_file = repo_root / "config" / "canonic_inventorry_methods_layers.json"

    if not executors_file.exists():
        print(f"ERROR: Executors file not found: {executors_file}")
        sys.exit(1)

    try:
        print(f"Identifying executors from {executors_file}...")
```

```
        inventory = generate_canonical_inventory(str(executors_file))

        print(f"Generated inventory with {len(inventory['methods'])} methods")

        for value in inventory["methods"].values():
            if any(
                key in str(value).lower() for key in ["score", "0.", "1.", "2.", "3."]
            ):
                if not isinstance(value, dict):
                    continue
                for k, v in value.items():
                    if k not in ["weights", "interaction_weights"] and isinstance(
                        v, int | float
                    ):
                        raise RuntimeError(
                            f"layer assignment corrupted: Found numeric score in JSON: {k}={v}"
                        )

        output_file.parent.mkdir(parents=True, exist_ok=True)

        with open(output_file, "w") as f:
            json.dump(inventory, f, indent=2, ensure_ascii=False)

        print(f"â\234\205 Successfully generated {output_file}")
        print(f"   Total executors: {inventory['_metadata']['total_executors']}")

    except RuntimeError as e:
        if "layer assignment corrupted" in str(e):
            print(f"â\235\214 ABORT: {e}")
            sys.exit(1)
        raise
    except Exception as e:
        print(f"â\235\214 ERROR: {e}")
        import traceback

        traceback.print_exc()
        sys.exit(1)


if __name__ == "__main__":
    main()



================================================================================
FILE: scripts/initialize_governance.py
================================================================================

#!/usr/bin/env python3
import os
import json
import hashlib
from pathlib import Path
import datetime

# Configuration
```

```
225: ROOT_DIR = Path(__file__).parent.parent
226: CONFIG_DIR = ROOT_DIR / "system" / "config"
227: HASH_REGISTRY = ROOT_DIR / "config_hash_registry.json"
228:
229: def compute_file_hash(filepath):
230:     sha256_hash = hashlib.sha256()
231:     with open(filepath, "rb") as f:
232:         # Read and update hash string value in blocks of 4K
233:         for byte_block in iter(lambda: f.read(4096), b""):
234:             sha256_hash.update(byte_block)
235:     return sha256_hash.hexdigest()
236:
237: def initialize_registry():
238:     print("Initializing Configuration Governance Registry...")
239:
240:     registry = {
241:         "_metadata": {
242:             "initialized_at": datetime.datetime.now(datetime.timezone.utc).isoformat(),
243:             "version": "1.0"
244:         },
245:         "files": {}
246:     }
247:
248:     if not CONFIG_DIR.exists():
249:         print(f"Config directory {CONFIG_DIR} does not exist. Creating...")
250:         CONFIG_DIR.mkdir(parents=True, exist_ok=True)
251:
252:     # Scan for config files
253:     for filepath in CONFIG_DIR.rglob("*"):
254:         if filepath.is_file() and filepath.name != ".DS_Store":
255:             file_hash = compute_file_hash(filepath)
256:             rel_path = str(filepath.relative_to(ROOT_DIR))
257:             registry["files"][rel_path] = {
258:                 "hash": file_hash,
259:                 "last_modified": datetime.datetime.fromtimestamp(filepath.stat().st_mtime).isoformat()
260:             }
261:             print(f"Registered: {rel_path}")
262:
263:     with open(HASH_REGISTRY, "w", encoding="utf-8") as f:
264:         json.dump(registry, f, indent=2)
265:
266:     print(f"Registry initialized at {HASH_REGISTRY}")
267:
268: if __name__ == "__main__":
269:     initialize_registry()
270:
271:
272:
273: ================================================================================
274: FILE: scripts/inventory/scan_methods_inventory.py
275: ================================================================================
276:
277: #!/usr/bin/env python3
278: """
279: Method Inventory Scanner
280:
```

```
Recursively scans src/farfan_pipeline/ to extract all top-level functions and
class methods (excluding nested inner functions). Captures module path, class name,
method name, signature details, line numbers, and builds canonical IDs.

Output format: List of method records with:
- module_path: Dot-separated module path (e.g., 'farfan_pipeline.core.orchestrator.core')
- class_name: Class name if method is in a class, else None
- method_name: Function/method name
- signature: Dict with parameters (name, default, type_hint, kind)
- line_number: Starting line number in source file
- canonical_id: Full identifier ('module.Class.method' or 'module.function')
- file_path: Relative path from repo root
"""

import ast
import json
import sys
from dataclasses import asdict, dataclass
from pathlib import Path


@dataclass
class MethodParameter:
    name: str
    kind: str
    default: str | None = None
    type_hint: str | None = None


@dataclass
class MethodRecord:
    module_path: str
    method_name: str
    signature: list[dict[str, str | None]]
    line_number: int
    canonical_id: str
    file_path: str
    class_name: str | None = None


class MethodInventoryScanner(ast.NodeVisitor):
    def __init__(self, module_path: str, file_path: str):
        self.module_path = module_path
        self.file_path = file_path
        self.methods: list[MethodRecord] = []
        self.current_class: str | None = None
        self.depth = 0

    def visit_ClassDef(self, node: ast.ClassDef) -> None:
        previous_class = self.current_class
        self.current_class = node.name
        self.generic_visit(node)
        self.current_class = previous_class

    def visit_FunctionDef(self, node: ast.FunctionDef) -> None:
        self._process_function(node)
```

```
337:
338:     def visit_AsyncFunctionDef(self, node: ast.AsyncFunctionDef) -> None:
339:         self._process_function(node)
340:
341:     def _process_function(self, node: ast.FunctionDef | ast.AsyncFunctionDef) -> None:
342:         if self.depth > 0:
343:             return
344:
345:         self.depth += 1
346:
347:         params = self._extract_parameters(node)
348:
349:         if self.current_class:
350:             canonical_id = f"{self.module_path}.{self.current_class}.{node.name}"
351:         else:
352:             canonical_id = f"{self.module_path}.{node.name}"
353:
354:         record = MethodRecord(
355:             module_path=self.module_path,
356:             class_name=self.current_class,
357:             method_name=node.name,
358:             signature=[asdict(p) for p in params],
359:             line_number=node.lineno,
360:             canonical_id=canonical_id,
361:             file_path=self.file_path,
362:         )
363:
364:         self.methods.append(record)
365:
366:         self.generic_visit(node)
367:         self.depth -= 1
368:
369:     def _extract_parameters(
370:         self, node: ast.FunctionDef | ast.AsyncFunctionDef
371:     ) -> list[MethodParameter]:
372:         params: list[MethodParameter] = []
373:         args = node.args
374:
375:         def get_type_hint(annotation: ast.expr | None) -> str | None:
376:             if annotation is None:
377:                 return None
378:             return ast.unparse(annotation)
379:
380:         def get_default(default: ast.expr | None) -> str | None:
381:             if default is None:
382:                 return None
383:             try:
384:                 return ast.unparse(default)
385:             except Exception:
386:                 return "<complex_default>"
387:
388:         all_args = args.posonlyargs + args.args
389:         defaults_offset = len(all_args) - len(args.defaults)
390:
391:         for i, arg in enumerate(all_args):
392:             default_value = None
```

```
            if i >= defaults_offset:
                default_value = get_default(args.defaults[i - defaults_offset])

            kind = "positional_or_keyword"
            if i < len(args.posonlyargs):
                kind = "positional_only"

            params.append(
                MethodParameter(
                    name=arg.arg,
                    kind=kind,
                    default=default_value,
                    type_hint=get_type_hint(arg.annotation),
                )
            )

        if args.vararg:
            params.append(
                MethodParameter(
                    name=args.vararg.arg,
                    kind="var_positional",
                    type_hint=get_type_hint(args.vararg.annotation),
                )
            )

        for arg in args.kwonlyargs:
            default_value = None
            for i, kw_arg in enumerate(args.kwonlyargs):
                if kw_arg.arg == arg.arg and i < len(args.kw_defaults):
                    default_value = get_default(args.kw_defaults[i])
                    break

            params.append(
                MethodParameter(
                    name=arg.arg,
                    kind="keyword_only",
                    default=default_value,
                    type_hint=get_type_hint(arg.annotation),
                )
            )

        if args.kwarg:
            params.append(
                MethodParameter(
                    name=args.kwarg.arg,
                    kind="var_keyword",
                    type_hint=get_type_hint(args.kwarg.annotation),
                )
            )

        return params


def scan_file(file_path: Path, repo_root: Path) -> list[MethodRecord]:
    relative_path = file_path.relative_to(repo_root)

```

```
    parts = relative_path.with_suffix("").parts
    if parts[0] == "src":
        parts = parts[1:]

    module_path = ".".join(parts)

    try:
        source = file_path.read_text(encoding="utf-8")
    except Exception as e:
        print(f"Warning: Could not read {file_path}: {e}", file=sys.stderr)
        return []

    try:
        tree = ast.parse(source, filename=str(file_path))
    except SyntaxError as e:
        print(f"Warning: Syntax error in {file_path}: {e}", file=sys.stderr)
        return []

    scanner = MethodInventoryScanner(module_path, str(relative_path))
    scanner.visit(tree)

    return scanner.methods


def scan_directory(directory: Path, repo_root: Path) -> list[MethodRecord]:
    all_methods: list[MethodRecord] = []

    python_files = sorted(directory.rglob("*.py"))

    for py_file in python_files:
        methods = scan_file(py_file, repo_root)
        all_methods.extend(methods)

    return all_methods


def main() -> None:
    repo_root = Path(__file__).resolve().parents[2]
    source_dir = repo_root / "src" / "farfan_pipeline"

    if not source_dir.exists():
        print(f"Error: Source directory not found: {source_dir}", file=sys.stderr)
        sys.exit(1)

    print(f"Scanning {source_dir}...", file=sys.stderr)

    methods = scan_directory(source_dir, repo_root)

    print(f"Found {len(methods)} methods", file=sys.stderr)

    output = {
        "metadata": {
            "total_methods": len(methods),
            "source_directory": str(source_dir.relative_to(repo_root)),
        },
        "methods": [asdict(m) for m in methods],
```

```
505:     }
506:
507:     print(json.dumps(output, indent=2))
508:
509:
510: if __name__ == "__main__":
511:     main()
512:
513:
514:
515: ================================================================================
516: FILE: scripts/inventory/verify_inventory.py
517: ================================================================================
518:
519: #!/usr/bin/env python3
520: """
521: Verification script for canonical_method_inventory.json
522:
523: This script validates the canonical method inventory against required criteria:
524: 1. Total methods >= 1995
525: 2. All 30 D1Q1-D6Q5 executors present with is_executor=true
526: 3. No duplicate canonical IDs
527: 4. Every method has non-empty role
528:
529: Exits with error code if any check fails and provides detailed diagnostic output.
530: """
531:
532: import json
533: import sys
534: from pathlib import Path
535: from typing import Any
536:
537: MIN_REQUIRED_METHODS = 1995
538: MAX_DISPLAY_ITEMS = 10
539:
540:
541: def load_inventory(inventory_path: Path) -> dict[str, Any]:  # type: ignore[misc]
542:     """Load and parse the canonical method inventory JSON file."""
543:     if not inventory_path.exists():
544:         print(f"â\235\214 ERROR: Inventory file not found: {inventory_path}")
545:         sys.exit(1)
546:
547:     try:
548:         with open(inventory_path, encoding="utf-8") as f:
549:             return json.load(f)  # type: ignore[no-any-return]
550:     except json.JSONDecodeError as e:
551:         print(f"â\235\214 ERROR: Failed to parse JSON: {e}")
552:         sys.exit(1)
553:     except Exception as e:
554:         print(f"â\235\214 ERROR: Failed to read file: {e}")
555:         sys.exit(1)
556:
557:
558: def check_total_methods(inventory: dict[str, Any]) -> tuple[bool, int]:  # type: ignore[misc]
559:     """Check if total_methods >= 1995."""
560:     metadata = inventory.get("metadata", {})
```

```
    total_methods = metadata.get("total_methods", 0)

    if isinstance(total_methods, str):
        try:
            total_methods = int(total_methods)
        except ValueError:
            total_methods = 0

    methods_dict = inventory.get("methods", {})
    actual_count = (
        len(methods_dict) if isinstance(methods_dict, dict) else len(methods_dict)
    )

    total_methods = max(total_methods, actual_count)

    passed = total_methods >= MIN_REQUIRED_METHODS
    return passed, total_methods


def generate_expected_executors() -> set[str]:
    """Generate the set of 30 expected executor identifiers (D1Q1-D6Q5)."""
    executors = set()
    for dim in range(1, 7):
        for question in range(1, 6):
            executors.add(f"D{dim}Q{question}")
    return executors


def check_executors(  # type: ignore[misc] # noqa: PLR0912
    inventory: dict[str, Any],
) -> tuple[bool, set[str], set[str]]:
    """Check if all 30 D1Q1-D6Q5 executors are present with is_executor=true."""
    expected_executors = generate_expected_executors()
    found_executors = set()

    methods = inventory.get("methods", {})

    if isinstance(methods, dict):
        for method_id, method_data in methods.items():
            if not isinstance(method_data, dict):
                continue

            is_executor = method_data.get("is_executor", False)
            if is_executor is True or str(is_executor).lower() == "true":
                search_fields = [
                    method_data.get("canonical_name", ""),
                    method_data.get("method_name", ""),
                    method_data.get("class_name", ""),
                    method_id,
                ]

                for executor_id in expected_executors:
                    patterns = [
                        executor_id,
                        executor_id.replace("Q", "_Q_"),
                        executor_id.replace("Q", "-Q"),
```

```
                        f"D{executor_id[1]}_Q{executor_id[3]}",
                    ]

                    for field in search_fields:
                        if field:
                            for pattern in patterns:
                                if pattern in field:
                                    found_executors.add(executor_id)
                                    break
    elif isinstance(methods, list):
        for method_data in methods:
            if not isinstance(method_data, dict):
                continue

            is_executor = method_data.get("is_executor", False)
            if is_executor is True or str(is_executor).lower() == "true":
                search_fields = [
                    method_data.get("canonical_name", ""),
                    method_data.get("method_name", ""),
                    method_data.get("class_name", ""),
                    method_data.get("method_id", ""),
                ]

                for executor_id in expected_executors:
                    patterns = [
                        executor_id,
                        executor_id.replace("Q", "_Q_"),
                        executor_id.replace("Q", "-Q"),
                        f"D{executor_id[1]}_Q{executor_id[3]}",
                    ]

                    for field in search_fields:
                        if field:
                            for pattern in patterns:
                                if pattern in field:
                                    found_executors.add(executor_id)
                                    break

    missing_executors = expected_executors - found_executors
    passed = len(missing_executors) == 0

    return passed, found_executors, missing_executors


def check_duplicate_ids(  # type: ignore[misc]
    inventory: dict[str, Any],
) -> tuple[bool, list[str]]:
    """Check for duplicate canonical identifiers."""
    methods = inventory.get("methods", {})

    seen_ids: set[str] = set()
    duplicates: list[str] = []

    if isinstance(methods, dict):
        for method_id, method_data in methods.items():
            if not isinstance(method_data, dict):
```

```
                continue

            canonical_id = method_data.get(
                "canonical_identifier",
                method_data.get("unique_id", method_data.get("method_id", method_id)),
            )

            if canonical_id in seen_ids:
                duplicates.append(canonical_id)
            else:
                seen_ids.add(canonical_id)
    elif isinstance(methods, list):
        for method_data in methods:
            if not isinstance(method_data, dict):
                continue

            canonical_id = method_data.get(
                "canonical_identifier",
                method_data.get("unique_id", method_data.get("method_id", "")),
            )

            if canonical_id in seen_ids:
                duplicates.append(canonical_id)
            else:
                seen_ids.add(canonical_id)

    passed = len(duplicates) == 0
    return passed, duplicates


def check_roles(inventory: dict[str, Any]) -> tuple[bool, list[str]]:  # type: ignore[misc]
    """Check that every method has a non-empty role."""
    methods = inventory.get("methods", {})
    methods_without_role: list[str] = []

    if isinstance(methods, dict):
        for method_id, method_data in methods.items():
            if not isinstance(method_data, dict):
                continue

            role = method_data.get("role", "")
            if not role or (isinstance(role, str) and role.strip() == ""):
                methods_without_role.append(method_id)
    elif isinstance(methods, list):
        for method_data in methods:
            if not isinstance(method_data, dict):
                continue

            method_id = method_data.get(
                "method_id",
                method_data.get(
                    "canonical_name", method_data.get("canonical_identifier", "UNKNOWN")
                ),
            )
            role = method_data.get("role", "")
            if not role or (isinstance(role, str) and role.strip() == ""):
```

```
729:                 methods_without_role.append(method_id)
730:
731:     passed = len(methods_without_role) == 0
732:     return passed, methods_without_role
733:
734:
735: def main() -> None:  # noqa: PLR0912, PLR0915
736:     """Main verification routine."""
737:     repo_root = Path(__file__).parent.parent.parent
738:     inventory_path = (
739:         repo_root / "scripts" / "inventory" / "canonical_method_inventory.json"
740:     )
741:
742:     print("=" * 80)
743:     print("CANONICAL METHOD INVENTORY VERIFICATION")
744:     print("=" * 80)
745:     print(f"\nInventory path: {inventory_path}")
746:     print()
747:
748:     inventory = load_inventory(inventory_path)
749:
750:     all_checks_passed = True
751:
752:     print("Running verification checks...\n")
753:
754:     # Check 1: Total methods >= 1995
755:     print("[1/4] Checking total methods count...")
756:     methods_check_passed, total_methods = check_total_methods(inventory)
757:     if methods_check_passed:
758:         print(f"â\234\205 PASS: Total methods = {total_methods} (>= {MIN_REQUIRED_METHODS})")
759:     else:
760:         print(
761:             f"â\235\214 FAIL: Total methods = {total_methods} (expected >= {MIN_REQUIRED_METHODS})"
762:         )
763:         print(f"   Deficit: {MIN_REQUIRED_METHODS - total_methods} methods")
764:         all_checks_passed = False
765:     print()
766:
767:     # Check 2: All 30 executors present
768:     print("[2/4] Checking for D1Q1-D6Q5 executors...")
769:     executors_check_passed, found_executors, missing_executors = check_executors(
770:         inventory
771:     )
772:     if executors_check_passed:
773:         print("â\234\205 PASS: All 30 executors present (D1Q1-D6Q5)")
774:         print(f"   Found executors: {sorted(found_executors)}")
775:     else:
776:         print(f"â\235\214 FAIL: Missing {len(missing_executors)} executor(s)")
777:         print(f"   Found: {len(found_executors)}/30 executors")
778:         print(f"   Found executors: {sorted(found_executors)}")
779:         print(f"   Missing executors: {sorted(missing_executors)}")
780:         all_checks_passed = False
781:     print()
782:
783:     # Check 3: No duplicate canonical IDs
784:     print("[3/4] Checking for duplicate canonical identifiers...")
```

```
    duplicates_check_passed, duplicates = check_duplicate_ids(inventory)
    if duplicates_check_passed:
        print("â\234\205 PASS: No duplicate canonical identifiers")
    else:
        print(f"â\235\214 FAIL: Found {len(duplicates)} duplicate identifier(s)")
        if len(duplicates) <= MAX_DISPLAY_ITEMS:
            for dup_id in duplicates:
                print(f"   - {dup_id}")
        else:
            for dup_id in duplicates[:MAX_DISPLAY_ITEMS]:
                print(f"   - {dup_id}")
            print(f"   ... and {len(duplicates) - MAX_DISPLAY_ITEMS} more")
        all_checks_passed = False
    print()

    # Check 4: All methods have non-empty roles
    print("[4/4] Checking that all methods have non-empty roles...")
    roles_check_passed, methods_without_role = check_roles(inventory)
    if roles_check_passed:
        print("â\234\205 PASS: All methods have non-empty roles")
    else:
        print(f"â\235\214 FAIL: Found {len(methods_without_role)} method(s) without role")
        if len(methods_without_role) <= MAX_DISPLAY_ITEMS:
            for method_id in methods_without_role:
                print(f"   - {method_id}")
        else:
            for method_id in methods_without_role[:MAX_DISPLAY_ITEMS]:
                print(f"   - {method_id}")
            print(f"   ... and {len(methods_without_role) - MAX_DISPLAY_ITEMS} more")
        all_checks_passed = False
    print()

    # Summary
    print("=" * 80)
    if all_checks_passed:
        print("â\234\205 ALL CHECKS PASSED")
        print("=" * 80)
        sys.exit(0)
    else:
        print("â\235\214 VERIFICATION FAILED")
        print("=" * 80)
        sys.exit(1)


if __name__ == "__main__":
    main()



================================================================================
FILE: scripts/mark_outdated_tests.py
================================================================================

#!/usr/bin/env python3
"""
Script to mark outdated tests with @pytest.mark.skip.
```

```
841:
842: Reads UPDATED_TESTS_MANIFEST.json and adds pytestmark skip to outdated test files.
843: """
844:
845: import json
846: import re
847: from pathlib import Path
848:
849:
850: def add_skip_marker_to_file(filepath: Path, reason: str) -> bool:
851:     """Add pytestmark skip to a test file."""
852:     if not filepath.exists():
853:         print(f"  â\232  File not found: {filepath}")
854:         return False
855:
856:     # Read current content
857:     content = filepath.read_text()
858:
859:     # Check if already marked
860:     if "pytestmark = pytest.mark.skip" in content:
861:         print(f"  â\234\223 Already marked: {filepath.name}")
862:         return True
863:
864:     # Find the import section
865:     lines = content.split('\n')
866:
867:     # Find where pytest is imported
868:     pytest_import_line = -1
869:     last_import_line = -1
870:     docstring_end = -1
871:
872:     in_docstring = False
873:     docstring_char = None
874:
875:     for i, line in enumerate(lines):
876:         stripped = line.strip()
877:
878:         # Track docstring
879:         if not in_docstring:
880:             if stripped.startswith('"""') or stripped.startswith("'''"):
881:                 in_docstring = True
882:                 docstring_char = stripped[:3]
883:                 # Check if it's a single-line docstring (starts and ends on same line)
884:                 if stripped.endswith(docstring_char) and len(stripped) > len(docstring_char):
885:                     # Single-line docstring
886:                     in_docstring = False
887:                     docstring_end = i
888:         else:
889:             if docstring_char in line:
890:                 in_docstring = False
891:                 docstring_end = i
892:
893:         # Track imports
894:         if stripped.startswith('import ') or stripped.startswith('from '):
895:             last_import_line = i
896:             if 'pytest' in stripped:
```

```
                pytest_import_line = i

    # Determine where to insert
    if pytest_import_line >= 0:
        insert_line = pytest_import_line + 1
    elif last_import_line >= 0:
        insert_line = last_import_line + 1
    elif docstring_end >= 0:
        insert_line = docstring_end + 1
    else:
        insert_line = 0

    # Skip empty lines after insertion point
    while insert_line < len(lines) and not lines[insert_line].strip():
        insert_line += 1

    # Insert the pytestmark
    marker_lines = [
        "",
        "# Mark all tests in this module as outdated",
        f'pytestmark = pytest.mark.skip(reason="{reason}")',
        ""
    ]

    # Insert marker
    lines = lines[:insert_line] + marker_lines + lines[insert_line:]

    # Write back
    filepath.write_text('\n'.join(lines))
    print(f"  â\234\223 Marked: {filepath.name}")
    return True


def main():
    """Mark all outdated tests."""
    print("=" * 70)
    print("Marking Outdated Tests")
    print("=" * 70)
    print()

    # Load manifest
    manifest_path = Path(__file__).parent.parent / "tests" / "UPDATED_TESTS_MANIFEST.json"
    if not manifest_path.exists():
        print(f"â\234\227 Manifest not found: {manifest_path}")
        return 1

    with open(manifest_path) as f:
        manifest = json.load(f)

    outdated_tests = manifest.get("outdated_tests", {}).get("tests", [])

    if not outdated_tests:
        print("â\234\223 No outdated tests to mark")
        return 0

    print(f"Found {len(outdated_tests)} outdated test files to mark\n")
```

```
    # Process each file
    repo_root = Path(__file__).parent.parent
    marked = 0
    skipped = 0

    for test_info in outdated_tests:
        filepath = repo_root / test_info["file"]
        reason = test_info["reason"]

        print(f"Processing: {test_info['file']}")
        if add_skip_marker_to_file(filepath, reason):
            marked += 1
        else:
            skipped += 1

    print()
    print("=" * 70)
    print(f"â\234\223 Marked: {marked} files")
    if skipped > 0:
        print(f"â\232  Skipped: {skipped} files")
    print("=" * 70)

    return 0


if __name__ == "__main__":
    import sys
    sys.exit(main())



================================================================================
FILE: scripts/operations/build_monolith.py
================================================================================

#!/usr/bin/env python3
"""
MonolithForge: Canonical Questionnaire Monolith Builder
=======================================================

Migrates legacy questionnaire.json and rubric_scoring.json into a single
questionnaire monolith with 305 questions (300 micro, 4 meso, 1 macro).

No graceful degradation. No strategic simplification. No atom loss.
Abort immediately on any inconsistency.
"""

import hashlib
import json
import logging
import sys
from collections import defaultdict
from dataclasses import dataclass
from datetime import datetime, timezone
from pathlib import Path
```

```
1009:
1010: from farfan_pipeline.core.orchestrator import get_questionnaire_provider
1011:
1012: QUESTIONNAIRE_PROVIDER = get_questionnaire_provider()
1013:
1014: # Configure structured logging
1015: logging.basicConfig(
1016:     level=logging.INFO,
1017:     format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
1018:     handlers=[
1019:         logging.FileHandler('forge.log'),
1020:         logging.StreamHandler()
1021:     ]
1022: )
1023: logger = logging.getLogger(__name__)
1024:
1025: class AbortError(Exception):
1026:     """Fatal error requiring immediate abort."""
1027:     def __init__(self, code: str, message: str, phase: str):
1028:         self.code = code
1029:         self.message = message
1030:         self.phase = phase
1031:         super().__init__(f"[{code}] {phase}: {message}")
1032:
1033: @dataclass
1034: class PhaseContext:
1035:     """Context for a construction phase."""
1036:     name: str
1037:     preconditions: list[str]
1038:     invariants: list[str]
1039:     postconditions: list[str]
1040:
1041: class MonolithForge:
1042:     """
1043:     Monolithic questionnaire builder following strict construction phases.
1044:     """
1045:
1046:     # Canonical constants
1047:     CANONICAL_POLICY_AREAS = ['P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9', 'P10']
1048:     CANONICAL_DIMENSIONS = ['D1', 'D2', 'D3', 'D4', 'D5', 'D6']
1049:     CANONICAL_SCORING_MODALITIES = ['TYPE_A', 'TYPE_B', 'TYPE_C', 'TYPE_D', 'TYPE_E', 'TYPE_F']
1050:
1051:     # Quality thresholds for micro questions
1052:     MICRO_QUALITY_LEVELS = {
1053:         'EXCELENTE': 0.85,
1054:         'BUENO': 0.70,
1055:         'ACEPTABLE': 0.55,
1056:         'INSUFICIENTE': 0.0
1057:     }
1058:
1059:     # Method sampling configuration
1060:     METHODS_PER_BASE_SLOT = 2  # Number of methods to sample from catalog per base_slot
1061:
1062:     def __init__(self):
1063:         self.legacy_data = {}
1064:         self.monolith = {}
```

```
        self.indices = {}
        self.stats = defaultdict(int)
        self.canonical_clusters = None  # Will be loaded from legacy data
        self.repo_root = Path(__file__).resolve().parents[1]  # Repository root

    def abort(self, code: str, message: str, phase: str):
        """Trigger immediate abort with error code."""
        logger.error(f"ABORT [{code}] in {phase}: {message}")
        raise AbortError(code, message, phase)

    # ========================================================================
    # PHASE 1: LoadLegacyPhase
    # ========================================================================

    def load_legacy_phase(self):
        """
        Load legacy JSON files with strict validation.
        Preconditions: Files exist, size > 0
        Invariants: Valid JSON, no null keys
        """
        phase = "LoadLegacyPhase"
        logger.info(f"=== {phase} START ===")

        # Whitelist of allowed files (relative to repo root)
        allowed_files = {
            'questionnaire.json': self.repo_root / 'questionnaire.json',
            'rubric_scoring.json': self.repo_root / 'rubric_scoring.json',
            'COMPLETE_METHOD_CLASS_MAP.json': self.repo_root / 'COMPLETE_METHOD_CLASS_MAP.json'
        }

        for name, path in allowed_files.items():

            # Precondition: file exists
            if not path.exists():
                self.abort('A001', f'Missing legacy file: {name}', phase)

            # Precondition: size > 0
            if path.stat().st_size == 0:
                self.abort('A001', f'Empty legacy file: {name}', phase)

            try:
                with open(path, encoding='utf-8') as f:
                    data = json.load(f)

                # Invariant: no null keys
                if None in data:
                    self.abort('A001', f'Null key in {name}', phase)

                self.legacy_data[name] = data
                logger.info(f"Loaded {name}: {len(str(data))} bytes")

            except json.JSONDecodeError as e:
                self.abort('A001', f'Invalid JSON in {name}: {e}', phase)

        # Load canonical clusters from questionnaire
        questionnaire = self.legacy_data['questionnaire.json']
```

```
        legacy_clusters = questionnaire.get('metadata', {}).get('clusters', [])
        self.canonical_clusters = {}
        legacy_to_canonical = {
            'P1': 'PA01',
            'P2': 'PA02',
            'P3': 'PA03',
            'P4': 'PA04',
            'P5': 'PA05',
            'P6': 'PA06',
            'P7': 'PA07',
            'P8': 'PA08',
            'P9': 'PA09',
            'P10': 'PA10',
        }

        for i, cluster_def in enumerate(legacy_clusters, 1):
            cluster_id = cluster_def.get('cluster_id') or f'CL{str(i).zfill(2)}'
            legacy_areas = cluster_def.get('legacy_policy_area_ids', [])
            canonical_areas = cluster_def.get('policy_area_ids', [])

            if not canonical_areas and legacy_areas:
                canonical_areas = [legacy_to_canonical.get(area, area) for area in legacy_areas]

            self.canonical_clusters[cluster_id] = {
                'canonical': canonical_areas,
                'legacy': legacy_areas,
            }

        logger.info(f"Loaded canonical clusters: {self.canonical_clusters}")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 2: StructuralIndexingPhase
    # ========================================================================

    def structural_indexing_phase(self):
        """
        Build indices for efficient lookup.
        Invariants: 300 micro questions, continuous numbering
        """
        phase = "StructuralIndexingPhase"
        logger.info(f"=== {phase} START ===")

        questionnaire = self.legacy_data['questionnaire.json']
        questions = questionnaire.get('questions', [])

        # Invariant: exactly 300 questions
        if len(questions) != 300:
            self.abort('A010', f'Expected 300 questions, found {len(questions)}', phase)

        # Build indices
        self.indices['by_global'] = {}
        self.indices['by_policy_area'] = defaultdict(list)
        self.indices['by_dimension'] = defaultdict(list)

        for q in questions:
```

```
            global_order = q.get('order', {}).get('global')

            if global_order is None:
                self.abort('A010', f"Question {q.get('question_id')} missing global order", phase)

            self.indices['by_global'][global_order] = q

            policy_area_id = q.get('policy_area_id')
            if policy_area_id:
                self.indices['by_policy_area'][policy_area_id].append(q)

            dimension_id = q.get('dimension_id')
            if dimension_id:
                self.indices['by_dimension'][dimension_id].append(q)

        # Invariant: continuous numbering
        expected_globals = set(range(1, 301))
        actual_globals = set(self.indices['by_global'].keys())

        if expected_globals != actual_globals:
            missing = expected_globals - actual_globals
            extra = actual_globals - expected_globals
            self.abort('A010', f'Discontinuous numbering. Missing: {missing}, Extra: {extra}', phase)

        logger.info(f"Indexed {len(questions)} questions")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 3: BaseSlotMappingPhase
    # ========================================================================

    def base_slot_mapping_phase(self):
        """
        Apply base_slot formula to all questions.
        Invariants: Each base_slot appears exactly 10 times
        """
        phase = "BaseSlotMappingPhase"
        logger.info(f"=== {phase} START ===")

        base_slot_counts = defaultdict(int)

        for global_num in range(1, 301):
            q = self.indices['by_global'][global_num]

            # Apply formula
            base_index = (global_num - 1) % 30
            base_slot = f"D{base_index//5+1}-Q{base_index%5+1}"

            # Store in question
            q['base_slot'] = base_slot
            q['question_global'] = global_num

            base_slot_counts[base_slot] += 1

        # Invariant: each base_slot exactly 10 times
        for slot, count in base_slot_counts.items():
```

```
1233:             if count != 10:
1234:                 self.abort('A020', f'Base slot {slot} has {count} instances, expected 10', phase)
1235:
1236:         # Verify we have exactly 30 base_slots
1237:         if len(base_slot_counts) != 30:
1238:             self.abort('A020', f'Expected 30 base_slots, found {len(base_slot_counts)}', phase)
1239:
1240:         logger.info(f"Mapped {len(base_slot_counts)} base_slots, each with 10 questions")
1241:         logger.info(f"=== {phase} COMPLETE ===")
1242:
1243:     # ========================================================================
1244:     # PHASE 4: ExtractionAndNormalizationPhase
1245:     # ========================================================================
1246:
1247:     def extraction_and_normalization_phase(self):
1248:         """
1249:         Normalize field by field with strict validation.
1250:         Invariants: text non-empty, scoring_modality valid
1251:         """
1252:         phase = "ExtractionAndNormalizationPhase"
1253:         logger.info(f"=== {phase} START ===")
1254:
1255:         for global_num in range(1, 301):
1256:             q = self.indices['by_global'][global_num]
1257:
1258:             # Normalize text (trim trailing whitespace, preserve internal)
1259:             text = q.get('question_text', '').strip()
1260:             if not text:
1261:                 self.abort('A030', f'Question {global_num} has empty text', phase)
1262:             q['text'] = text
1263:
1264:             # Validate scoring_modality
1265:             scoring_modality = q.get('scoring_modality')
1266:             if scoring_modality not in self.CANONICAL_SCORING_MODALITIES:
1267:                 self.abort('A030', f'Question {global_num} has invalid scoring_modality: {scoring_modality}', phase)
1268:
1269:             # Ensure no FIXME/TODO/TEMP markers
1270:             for marker in ['FIXME', 'TODO', 'TEMP', 'LEGACY']:
1271:                 if marker in str(q):
1272:                     self.abort('A030', f'Question {global_num} contains forbidden marker: {marker}', phase)
1273:
1274:         logger.info("Normalized 300 micro questions")
1275:         logger.info(f"=== {phase} COMPLETE ===")
1276:
1277:     # ========================================================================
1278:     # PHASE 5: IndicatorsAndEvidencePhase
1279:     # ========================================================================
1280:
1281:     def indicators_and_evidence_phase(self):
1282:         """
1283:         Separate expected_elements, patterns, validation_checks.
1284:         Invariants: Exact count preservation, no silent duplication
1285:         """
1286:         phase = "IndicatorsAndEvidencePhase"
1287:         logger.info(f"=== {phase} START ===")
1288:
```

```
        for global_num in range(1, 301):
            q = self.indices['by_global'][global_num]

            # Extract expected_elements from evidence_expectations structure
            evidence_exp = q.get('evidence_expectations', {})

            # Build expected_elements from the evidence expectations
            expected_elements = []
            for key, value in evidence_exp.items():
                if key.endswith('_minimos') or key.endswith('_minimas'):
                    # Extract minimum requirements
                    expected_elements.append({
                        'type': key.replace('_minimos', '').replace('_minimas', ''),
                        'minimum': value
                    })
                elif isinstance(value, bool) and value:
                    # Boolean requirements
                    expected_elements.append({
                        'type': key,
                        'required': True
                    })

            # If no elements extracted, use the required_evidence_keys
            if not expected_elements:
                req_keys = q.get('required_evidence_keys', [])
                expected_elements = [{'type': key} for key in req_keys] if req_keys else []

            # Still nothing? Extract from validation_checks
            if not expected_elements:
                validation_checks = q.get('validation_checks', {})
                if validation_checks:
                    expected_elements = [
                        {'type': check_name, 'minimum': check_data.get('minimum_required', 1)}
                        for check_name, check_data in validation_checks.items()
                        if isinstance(check_data, dict)
                    ]

            # Abort if still no elements
            if not expected_elements:
                self.abort('A040', f'Question {global_num} missing expected_elements', phase)

            q['expected_elements'] = expected_elements

            # Extract patterns
            patterns = q.get('search_patterns', [])
            if isinstance(patterns, dict):
                # Flatten patterns from different structures
                pattern_list = []
                for key, val in patterns.items():
                    if isinstance(val, list):
                        pattern_list.extend(val)
                    else:
                        pattern_list.append(val)
                patterns = pattern_list

            q['pattern_refs'] = patterns if patterns else []
```

```
1345:
1346:             # Extract validation_checks
1347:             validations = q.get('validation_checks', {})
1348:             if isinstance(validations, dict):
1349:                 # Keep as structured dictionary
1350:                 q['validations'] = validations
1351:             else:
1352:                 q['validations'] = validations if validations else {}
1353:
1354:         logger.info("Extracted indicators and evidence for 300 questions")
1355:         logger.info(f"=== {phase} COMPLETE ===")
1356:
1357:     # ========================================================================
1358:     # PHASE 6: MethodSetSynthesisPhase
1359:     # ========================================================================
1360:
1361:     def method_set_synthesis_phase(self):
1362:         """
1363:         Insert method_sets per base_slot from method catalog.
1364:         Invariants: Each method has class, function, description, priority 1-3
1365:         """
1366:         phase = "MethodSetSynthesisPhase"
1367:         logger.info(f"=== {phase} START ===")
1368:
1369:         # Load real method catalog from canonical source
1370:         from farfan_pipeline.core.orchestrator.factory import load_catalog
1371:
1372:         # Load canonical method catalog (1,996 methods)
1373:         catalog_path = self.repo_root / "config" / "canonical_method_catalog.json"
1374:         try:
1375:             catalog = load_catalog()
1376:             logger.info(f"Loaded method catalog with {catalog.get('summary', {}).get('total_methods', 0)} methods")
1377:         except FileNotFoundError as e:
1378:             self.abort('A050', f'Method catalog not found at {catalog_path}: {e}', phase)
1379:         except Exception as e:
1380:             self.abort(
1381:                 'A050',
1382:                 f'Failed to load method catalog: {e}\n'
1383:                 f'Check that the catalog file exists at {catalog_path} and is in the expected format.',
1384:                 phase
1385:             )
1386:
1387:         # Extract methods from catalog and organize by base_slot
1388:         base_slot_methods = {}
1389:
1390:         # The catalog structure has files with methods
1391:         files_data = catalog.get('files', {})
1392:
1393:         # Create method sets per base_slot
1394:         # Since the catalog doesn't directly map to base_slots, we create a
1395:         # reasonable distribution of methods across base_slots
1396:         for d_num in range(1, 7):  # D1-D6
1397:             for q_num in range(1, 6):  # Q1-Q5
1398:                 base_slot = f"D{d_num}-Q{q_num}"
1399:
1400:                 # Assign methods from catalog to this base_slot
```

```
                # This mapping should ideally come from configuration
                base_slot_methods[base_slot] = self._get_methods_for_base_slot(
                    base_slot, files_data
                )

        # Apply to questions
        for global_num in range(1, 301):
            q = self.indices['by_global'][global_num]
            base_slot = q['base_slot']

            methods = base_slot_methods.get(base_slot, [])

            # Invariant: methods have required fields
            for method in methods:
                if not method.get('description'):
                    self.abort('A050', f'Method for {base_slot} missing description', phase)
                if method.get('priority') not in [1, 2, 3]:
                    self.abort('A050', f'Method for {base_slot} has invalid priority: {method.get("priority")}', phase)

            q['method_sets'] = methods

        logger.info(f"Applied method sets from catalog to 30 base_slots")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 7: RubricTranspositionPhase
    # ========================================================================

    def rubric_transposition_phase(self):
        """
        Transfer qualitative levels and modalities from rubric.
        Invariants: min_score descending order
        """
        phase = "RubricTranspositionPhase"
        logger.info(f"=== {phase} START ===")

        rubric = self.legacy_data['rubric_scoring.json']

        # Extract scoring modalities
        scoring_modalities = rubric.get('scoring_modalities', {})

        # Build scoring_matrix for micro questions
        scoring_matrix = {}

        for modality_key in self.CANONICAL_SCORING_MODALITIES:
            if modality_key in scoring_modalities:
                modality_info = scoring_modalities[modality_key]
                scoring_matrix[modality_key] = {
                    'description': modality_info.get('description', ''),
                    'max_score': modality_info.get('max_score', 3),
                    'levels': []
                }

        # Create micro quality levels
        micro_levels = []
        prev_score = float('inf')
```

```
1457:
1458:         for level_name, min_score in sorted(self.MICRO_QUALITY_LEVELS.items(), key=lambda x: -x[1]):
1459:             # Invariant: descending order
1460:             if min_score >= prev_score:
1461:                 self.abort('A060', f'Rubric thresholds not descending: {level_name}', phase)
1462:
1463:             micro_levels.append({
1464:                 'level': level_name,
1465:                 'min_score': min_score,
1466:                 'color': self._get_level_color(level_name)
1467:             })
1468:             prev_score = min_score
1469:
1470:         self.monolith['scoring_matrix'] = {
1471:             'micro_levels': micro_levels,
1472:             'modalities': scoring_matrix
1473:         }
1474:
1475:         logger.info(f"Transposed rubric with {len(micro_levels)} quality levels")
1476:         logger.info(f"=== {phase} COMPLETE ===")
1477:
1478:     def _get_level_color(self, level_name: str) -> str:
1479:         """Map quality level to color."""
1480:         colors = {
1481:             'EXCELENTE': 'green',
1482:             'BUENO': 'blue',
1483:             'ACEPTABLE': 'yellow',
1484:             'INSUFICIENTE': 'red'
1485:         }
1486:         return colors.get(level_name, 'gray')
1487:
1488:     # ========================================================================
1489:     # PHASE 8: MesoMacroEmbeddingPhase
1490:     # ========================================================================
1491:
1492:     def meso_macro_embedding_phase(self):
1493:         """
1494:         Insert 4 MESO cluster questions and 1 MACRO question.
1495:         Invariants: Clusters EXACT, hermeticity preserved
1496:         """
1497:         phase = "MesoMacroEmbeddingPhase"
1498:         logger.info(f"=== {phase} START ===")
1499:
1500:         # Verify cluster hermeticity BEFORE insertion
1501:         questionnaire = self.legacy_data['questionnaire.json']
1502:         legacy_clusters = questionnaire.get('metadata', {}).get('clusters', [])
1503:
1504:         # Map legacy clusters to canonical
1505:         cluster_mapping = {}
1506:         for i, cluster_def in enumerate(legacy_clusters, 1):
1507:             cluster_id = cluster_def.get('cluster_id') or f"CL{str(i).zfill(2)}"
1508:             legacy_areas = cluster_def.get('legacy_policy_area_ids', [])
1509:             canonical_record = self.canonical_clusters.get(cluster_id)
1510:
1511:             if not canonical_record:
1512:                 self.abort('A070', f'Cluster {cluster_id} not found in canonical registry', phase)
```

```
1513:
1514:             canonical_areas = canonical_record.get('canonical', [])
1515:             expected_legacy = canonical_record.get('legacy', [])
1516:
1517:             if expected_legacy and set(legacy_areas) != set(expected_legacy):
1518:                 self.abort(
1519:                     'A070',
1520:                     f'{cluster_id} legacy hermeticity violation. '
1521:                     f'Expected {expected_legacy}, got {legacy_areas}',
1522:                     phase
1523:                 )
1524:
1525:             cluster_mapping[cluster_id] = {
1526:                 'cluster_id': cluster_id,
1527:                 'policy_area_ids': canonical_areas,
1528:                 'legacy_policy_area_ids': legacy_areas,
1529:                 'label_es': cluster_def.get('i18n', {}).get('keys', {}).get('label_es', ''),
1530:                 'label_en': cluster_def.get('i18n', {}).get('keys', {}).get('label_en', ''),
1531:                 'rationale': cluster_def.get('rationale', '')
1532:             }
1533:
1534:         # Create 4 MESO questions (one per cluster)
1535:         meso_questions = []
1536:
1537:         for i, (cluster_id, cluster_info) in enumerate(sorted(cluster_mapping.items()), 301):
1538:             meso_q = {
1539:                 'question_global': i,
1540:                 'question_id': f'MESO_{i-300}',
1541:                 'cluster_id': cluster_id,
1542:                 'type': 'MESO',
1543:                 'text': f"Â¿CÃ³mo se integran las polÃticas en el cluster {cluster_info['label_es']}?",
1544:                 'policy_areas': cluster_info['policy_area_ids'],
1545:                 'scoring_modality': 'MESO_INTEGRATION',
1546:                 'aggregation_method': 'weighted_average',
1547:                 'patterns': [
1548:                     {
1549:                         'type': 'cross_reference',
1550:                         'description': f'Verificar referencias cruzadas entre Ã¡reas {cluster_info["policy_area_ids"]}'
1551:                     },
1552:                     {
1553:                         'type': 'coherence',
1554:                         'description': 'Evaluar coherencia narrativa entre polÃticas del cluster'
1555:                     }
1556:                 ]
1557:             }
1558:             meso_questions.append(meso_q)
1559:
1560:         # Create 1 MACRO question
1561:         macro_question = {
1562:             'question_global': 305,
1563:             'question_id': 'MACRO_1',
1564:             'type': 'MACRO',
1565:             'text': 'Â¿El Plan de Desarrollo presenta una visiÃ³n integral y coherente que articula todos los clusters y dimensiones?',
1566:             'scoring_modality': 'MACRO_HOLISTIC',
1567:             'aggregation_method': 'holistic_assessment',
1568:             'clusters': list(self.canonical_clusters.keys()),
```

```
            'patterns': [
                {
                    'type': 'narrative_coherence',
                    'description': 'Evaluar coherencia narrativa global del plan',
                    'priority': 1
                },
                {
                    'type': 'cross_cluster_integration',
                    'description': 'Verificar integraciÃ³n entre todos los clusters',
                    'priority': 1
                },
                {
                    'type': 'long_term_vision',
                    'description': 'Evaluar visiÃ³n de largo plazo y transformaciÃ³n estructural',
                    'priority': 2
                }
            ],
            'fallback': {
                'pattern': 'MACRO_AMBIGUO',
                'condition': 'always_true',
                'priority': 999
            }
        }

        # Store in monolith
        self.monolith['meso_questions'] = meso_questions
        self.monolith['macro_question'] = macro_question

        logger.info(f"Embedded {len(meso_questions)} MESO + 1 MACRO questions")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 9: IntegritySealingPhase
    # ========================================================================

    def integrity_sealing_phase(self):
        """
        Calculate monolith hash for integrity verification.
        Postconditions: hash is reproducible
        """
        phase = "IntegritySealingPhase"
        logger.info(f"=== {phase} START ===")

        # Build final monolith structure
        monolith = {
            'schema_version': '1.1.0',  # Added versioning
            'version': '1.0.0',
            'generated_at': datetime.now(timezone.utc).isoformat(),
            'blocks': {}
        }

        # Block A: niveles_abstraccion
        questionnaire = self.legacy_data['questionnaire.json']
        monolith['blocks']['niveles_abstraccion'] = {
            'policy_areas': questionnaire['metadata'].get('policy_areas', []),
            'dimensions': questionnaire['metadata'].get('dimensions', []),
```

```
            'clusters': questionnaire['metadata'].get('clusters', [])
        }

        # Block B: micro_questions (300)
        micro_questions = []
        for global_num in range(1, 301):
            q = self.indices['by_global'][global_num]

            # Structure patterns with categories
            structured_patterns = self._structure_patterns(q['pattern_refs'], q['question_id'])

            micro_questions.append({
                'question_global': q['question_global'],
                'question_id': q.get('question_id'),
                'base_slot': q['base_slot'],
                'policy_area_id': q.get('policy_area_id'),
                'dimension_id': q.get('dimension_id'),
                'cluster_id': q.get('cluster_id'),
                'text': q['text'],
                'scoring_modality': q['scoring_modality'],
                'scoring_definition_ref': f"scoring_modalities.{q['scoring_modality']}",
                'expected_elements': q['expected_elements'],
                'patterns': structured_patterns,  # Enhanced structure
                'validations': q['validations'],
                'method_sets': q['method_sets'],
                'failure_contract': {
                    'abort_if': ['missing_required_element', 'incomplete_text'],
                    'emit_code': f"ABORT-{q.get('question_id')}-REQ"
                }
            })

        monolith['blocks']['micro_questions'] = micro_questions

        # Block C: meso_questions (4)
        monolith['blocks']['meso_questions'] = self.monolith['meso_questions']

        # Block D: macro_question (1)
        monolith['blocks']['macro_question'] = self.monolith['macro_question']

        # Add scoring matrix with explicit definitions
        monolith['blocks']['scoring'] = self._create_scoring_definitions()

        # Add semantic layers block
        monolith['blocks']['semantic_layers'] = {
            'embedding_strategy': {
                'model': 'multilingual-e5-base',
                'dimension': 768,
                'hybrid': {
                    'bm25': True,
                    'fusion': 'RRF'
                }
            },
            'disambiguation': {
                'entity_linker': 'spaCy_es_core_news_lg',
                'confidence_threshold': 0.72
            }
```

```
        }

        # Add observability block
        monolith['observability'] = {
            'telemetry_schema': {
                'metrics': [
                    {
                        'name': 'pattern_match_count',
                        'level': 'MICRO',
                        'aggregation': 'sum'
                    },
                    {
                        'name': 'rule_latency_ms',
                        'level': 'METHOD_SET',
                        'aggregation': 'p95'
                    },
                    {
                        'name': 'validation_failure_rate',
                        'level': 'DIMENSION',
                        'aggregation': 'ratio'
                    }
                ],
                'logs': {
                    'format': 'jsonl',
                    'fields': ['timestamp', 'question_id', 'pattern_id', 'matched_text', 'confidence', 'trace_id', 'ruleset_hash']
                },
                'tracing': {
                    'propagation': 'W3C',
                    'span_structure': ['LOAD_RULESET', 'PARSE_DOCUMENT', 'EXTRACT_PATTERN', 'VALIDATE', 'AGGREGATE', 'EMIT_SCORE']
                }
            }
        }

        # Calculate ruleset hash for deterministic reproducibility
        ruleset_hash = self._calculate_ruleset_hash(micro_questions)

        # Calculate hash on canonical serialization
        canonical_json = json.dumps(monolith, sort_keys=True, ensure_ascii=False, separators=(',', ':'))
        monolith_hash = hashlib.sha256(canonical_json.encode('utf-8')).hexdigest()

        # Add integrity block
        monolith['integrity'] = {
            'monolith_hash': monolith_hash,
            'ruleset_hash': ruleset_hash,
            'question_count': {
                'micro': 300,
                'meso': 4,
                'macro': 1,
                'total': 305
            }
        }

        self.monolith['final'] = monolith

        logger.info(f"Sealed monolith with hash: {monolith_hash[:16]}...")
        logger.info(f"Ruleset hash: {ruleset_hash[:16]}...")
```

```
1737:         logger.info(f"=== {phase} COMPLETE ===")
1738:
1739:     def _structure_patterns(self, pattern_refs: list, question_id: str) -> list:
1740:         """Structure pattern_refs as typed objects with categories."""
1741:         structured = []
1742:
1743:         for idx, pattern in enumerate(pattern_refs):
1744:             if not isinstance(pattern, str):
1745:                 continue
1746:
1747:             # Categorize patterns based on content
1748:             category = self._categorize_pattern(pattern)
1749:
1750:             structured.append({
1751:                 'id': f"PAT-{question_id}-{idx:03d}",
1752:                 'pattern': pattern,
1753:                 'category': category,
1754:                 'match_type': 'REGEX' if any(c in pattern for c in r'\\d.*+?[]()') else 'LITERAL',
1755:                 'flags': 'i',
1756:                 'confidence_weight': 0.85
1757:             })
1758:
1759:         return structured
1760:
1761:     def _categorize_pattern(self, pattern: str) -> str:
1762:         """Categorize a pattern based on its content."""
1763:         pattern_lower = pattern.lower()
1764:
1765:         if any(src in pattern_lower for src in ['dane', 'medicina legal', 'fiscalÃ-a', 'policÃ-a', 'sivigila', 'sispro']):
1766:             return 'FUENTE_OFICIAL'
1767:         elif any(ind in pattern_lower for ind in ['tasa', 'porcentaje', '%', 'indicador', 'cifra']):
1768:             return 'INDICADOR'
1769:         elif any(year in pattern for year in ['20\\d{2}', 'aÃ±o', 'periodo']):
1770:             return 'TEMPORAL'
1771:         elif any(ent in pattern_lower for ent in ['departamental', 'municipal', 'territorial']):
1772:             return 'TERRITORIAL'
1773:         elif any(unit in pattern_lower for unit in ['por 100', 'por 1.000', 'por cada']):
1774:             return 'UNIDAD_MEDIDA'
1775:         else:
1776:             return 'GENERAL'
1777:
1778:     def _create_scoring_definitions(self):
1779:         """Create explicit scoring modality definitions."""
1780:         return {
1781:             'micro_levels': self.monolith['scoring_matrix']['micro_levels'],
1782:             'modalities': self.monolith['scoring_matrix']['modalities'],
1783:             'modality_definitions': {
1784:                 'TYPE_A': {
1785:                     'description': 'Count 4 elements and scale to 0-3',
1786:                     'aggregation': 'presence_threshold',
1787:                     'threshold': 0.7,
1788:                     'failure_code': 'F-A-MIN'
1789:                 },
1790:                 'TYPE_B': {
1791:                     'description': 'Count up to 3 elements, each worth 1 point',
1792:                     'aggregation': 'binary_sum',
```

```
1793:                     'max_score': 3,
1794:                     'failure_code': 'F-B-MIN'
1795:                 },
1796:                 'TYPE_C': {
1797:                     'description': 'Count 2 elements and scale to 0-3',
1798:                     'aggregation': 'presence_threshold',
1799:                     'threshold': 0.5,
1800:                     'failure_code': 'F-C-MIN'
1801:                 },
1802:                 'TYPE_D': {
1803:                     'description': 'Count 3 elements, weighted',
1804:                     'aggregation': 'weighted_sum',
1805:                     'weights': [0.4, 0.3, 0.3],
1806:                     'failure_code': 'F-D-MIN'
1807:                 },
1808:                 'TYPE_E': {
1809:                     'description': 'Boolean presence check',
1810:                     'aggregation': 'binary_presence',
1811:                     'failure_code': 'F-E-MIN'
1812:                 },
1813:                 'TYPE_F': {
1814:                     'description': 'Continuous scale',
1815:                     'aggregation': 'normalized_continuous',
1816:                     'normalization': 'minmax',
1817:                     'failure_code': 'F-F-MIN'
1818:                 }
1819:             }
1820:         }
1821:
1822:     def _calculate_ruleset_hash(self, micro_questions: list) -> str:
1823:         """Calculate deterministic hash of all patterns for reproducibility."""
1824:         all_patterns = []
1825:
1826:         for q in micro_questions:
1827:             for pattern in q.get('patterns', []):
1828:                 if isinstance(pattern, dict):
1829:                     all_patterns.append(pattern.get('pattern', ''))
1830:                 else:
1831:                     all_patterns.append(str(pattern))
1832:
1833:         # Sort for determinism
1834:         all_patterns.sort()
1835:
1836:         # Concatenate and hash
1837:         patterns_str = '|'.join(all_patterns)
1838:         return hashlib.sha256(patterns_str.encode('utf-8')).hexdigest()
1839:
1840:     def _map_priority(self, priority_str: str) -> int:
1841:         """Map string priority to numeric priority (1-3)."""
1842:         priority_map = {
1843:             'CRITICAL': 1,
1844:             'HIGH': 1,
1845:             'MEDIUM': 2,
1846:             'LOW': 3,
1847:         }
1848:         return priority_map.get(priority_str, 2)
```

```
1849:
1850:     def _get_methods_for_base_slot(self, base_slot: str, files_data: dict) -> list[dict]:
1851:         """Get methods for a specific base_slot from catalog data.
1852:
1853:         Args:
1854:             base_slot: Base slot identifier (e.g., 'D1-Q1')
1855:             files_data: Method catalog files data
1856:
1857:         Returns:
1858:             List of method definitions for this base_slot
1859:         """
1860:         # Extract dimension and question from base_slot
1861:         # This is a simplified mapping - production code should use proper configuration
1862:         methods = []
1863:
1864:         # Get a subset of methods from the catalog for this base_slot
1865:         # In a real implementation, this mapping should come from configuration
1866:         for file_name, file_data in files_data.items():
1867:             file_methods = file_data.get('methods', [])
1868:
1869:             # Take first METHODS_PER_BASE_SLOT methods as a representative sample
1870:             for method_info in file_methods[:self.METHODS_PER_BASE_SLOT]:
1871:                 methods.append({
1872:                     'class': method_info.get('class', 'UnknownClass'),
1873:                     'function': method_info.get('method_name', 'unknown_method'),
1874:                     'module': file_name,
1875:                     'method_type': 'analysis',
1876:                     'priority': self._map_priority(method_info.get('priority', 'MEDIUM')),
1877:                     'description': method_info.get('description', f"Analysis method for {base_slot}")
1878:                 })
1879:
1880:             # Only need a few methods per base_slot
1881:             if len(methods) >= self.METHODS_PER_BASE_SLOT:
1882:                 break
1883:
1884:         # Ensure we have at least one method
1885:         if not methods:
1886:             logger.warning(
1887:                 f"No analysis methods found in catalog for base_slot '{base_slot}'. "
1888:                 f"Using synthetic DefaultAnalyzer fallback. This may indicate a configuration issue."
1889:             )
1890:             methods.append({
1891:                 'class': 'DefaultAnalyzer',
1892:                 'function': 'analyze',
1893:                 'module': 'default',
1894:                 'method_type': 'analysis',
1895:                 'priority': 2,
1896:                 'description': f'Default analysis for {base_slot}'
1897:             })
1898:
1899:         return methods
1900:
1901:     # ========================================================================
1902:     # PHASE 10: ValidationReportPhase
1903:     # ========================================================================
1904:
```

```
1905:     def validation_report_phase(self):
1906:         """
1907:         Validate all invariants before emission.
1908:         Invariants: All legacy counters == destination counters
1909:         """
1910:         phase = "ValidationReportPhase"
1911:         logger.info(f"=== {phase} START ===")
1912:
1913:         monolith = self.monolith['final']
1914:
1915:         # Validate question counts
1916:         micro_count = len(monolith['blocks']['micro_questions'])
1917:         meso_count = len(monolith['blocks']['meso_questions'])
1918:         macro_count = 1
1919:         total_count = micro_count + meso_count + macro_count
1920:
1921:         if micro_count != 300:
1922:             self.abort('A090', f'Expected 300 micro questions, got {micro_count}', phase)
1923:
1924:         if meso_count != 4:
1925:             self.abort('A090', f'Expected 4 meso questions, got {meso_count}', phase)
1926:
1927:         if total_count != 305:
1928:             self.abort('A090', f'Expected 305 total questions, got {total_count}', phase)
1929:
1930:         # Validate base_slot coverage
1931:         base_slot_counts = defaultdict(int)
1932:         for q in monolith['blocks']['micro_questions']:
1933:             base_slot_counts[q['base_slot']] += 1
1934:
1935:         for slot, count in base_slot_counts.items():
1936:             if count != 10:
1937:                 self.abort('A090', f'Base slot {slot} has {count} questions, expected 10', phase)
1938:
1939:         if len(base_slot_counts) != 30:
1940:             self.abort('A090', f'Expected 30 base_slots, got {len(base_slot_counts)}', phase)
1941:
1942:         # Validate cluster hermeticity
1943:         clusters_in_monolith = monolith['blocks']['niveles_abstraccion']['clusters']
1944:         for cluster_def in clusters_in_monolith:
1945:             cluster_id = cluster_def.get('cluster_id')
1946:             canonical_record = self.canonical_clusters.get(cluster_id)
1947:
1948:             if not canonical_record:
1949:                 self.abort('A090', f'Cluster {cluster_id} missing from canonical registry', phase)
1950:
1951:             canonical_expected = set(canonical_record.get('canonical', []))
1952:             canonical_present = set(cluster_def.get('policy_area_ids', []))
1953:             if canonical_expected and canonical_present != canonical_expected:
1954:                 self.abort(
1955:                     'A090',
1956:                     f'{cluster_id} canonical hermeticity violation in final',
1957:                     phase
1958:                 )
1959:
1960:             expected_legacy = set(canonical_record.get('legacy', []))
```

```
            legacy_present = set(cluster_def.get('legacy_policy_area_ids', []))
            if expected_legacy and legacy_present != expected_legacy:
                self.abort(
                    'A090',
                    f'{cluster_id} legacy hermeticity violation in final',
                    phase
                )

        logger.info("Validation PASSED:")
        logger.info("  - 300 micro questions")
        logger.info("  - 4 meso questions")
        logger.info("  - 1 macro question")
        logger.info("  - 30 base_slots, each with 10 questions")
        logger.info("  - Cluster hermeticity verified")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 11: FinalEmissionPhase
    # ========================================================================

    def final_emission_phase(self, output_path: str | None):
        """Write the orchestrator-managed questionnaire monolith to disk."""
        phase = "FinalEmissionPhase"
        logger.info(f"=== {phase} START ===")

        monolith = self.monolith['final']

        # Delegate persistence to the orchestrator provider
        output_file = QUESTIONNAIRE_PROVIDER.save(monolith, output_path=output_path)

        file_info = QUESTIONNAIRE_PROVIDER.describe(output_file)
        file_size = file_info["size"]
        if file_size == 0:
            self.abort('A100', 'Empty monolith emission', phase)

        # Verify hash matches by reloading through the provider interface
        reloaded = QUESTIONNAIRE_PROVIDER.load(force_reload=True, data_path=output_file)

        expected_hash = monolith['integrity']['monolith_hash']
        reloaded_without_integrity = {k: v for k, v in reloaded.items() if k != 'integrity'}
        canonical_check = json.dumps(reloaded_without_integrity, sort_keys=True, ensure_ascii=False, separators=(',', ':'))
        actual_hash = hashlib.sha256(canonical_check.encode('utf-8')).hexdigest()

        if actual_hash != expected_hash:
            self.abort('A080', f'Hash mismatch after emission: expected {expected_hash}, got {actual_hash}', phase)

        logger.info(f"Emitted monolith to {output_path}")
        logger.info(f"  File size: {file_size:,} bytes")
        logger.info(f"  Hash: {expected_hash[:16]}... (verified)")
        logger.info(f"=== {phase} COMPLETE ===")

        # Generate manifest
        manifest = {
            'timestamp': datetime.now(timezone.utc).isoformat(),
            'output_file': str(output_file),
            'file_size': file_size,
```

```
            'monolith_hash': expected_hash,
            'phases_executed': [
                'LoadLegacyPhase',
                'StructuralIndexingPhase',
                'BaseSlotMappingPhase',
                'ExtractionAndNormalizationPhase',
                'IndicatorsAndEvidencePhase',
                'MethodSetSynthesisPhase',
                'RubricTranspositionPhase',
                'MesoMacroEmbeddingPhase',
                'IntegritySealingPhase',
                'ValidationReportPhase',
                'FinalEmissionPhase'
            ],
            'stats': {
                'micro_questions': 300,
                'meso_questions': 4,
                'macro_questions': 1,
                'total_questions': 305,
                'base_slots': 30,
                'clusters': 4,
                'policy_areas': 10,
                'dimensions': 6
            }
        }

        manifest_path = output_file.parent / 'forge_manifest.json'
        with open(manifest_path, 'w', encoding='utf-8') as f:
            json.dump(manifest, f, indent=2, ensure_ascii=False)

        logger.info(f"Manifest written to {manifest_path}")

    # ========================================================================
    # Main Build Pipeline
    # ========================================================================

    def build(self, output_path: str | None = None):
        """Execute all construction phases in order."""
        logger.info("=" * 70)
        logger.info("MonolithForge: Starting construction pipeline")
        logger.info("=" * 70)

        try:
            self.load_legacy_phase()
            self.structural_indexing_phase()
            self.base_slot_mapping_phase()
            self.extraction_and_normalization_phase()
            self.indicators_and_evidence_phase()
            self.method_set_synthesis_phase()
            self.rubric_transposition_phase()
            self.meso_macro_embedding_phase()
            self.integrity_sealing_phase()
            self.validation_report_phase()
            self.final_emission_phase(output_path)

            logger.info("=" * 70)
```

```
            logger.info("MonolithForge: Construction COMPLETE")
            logger.info("=" * 70)
            return True

        except AbortError as e:
            logger.error(f"FATAL ABORT: {e}")
            logger.error(f"Construction FAILED at {e.phase}")
            return False

def main():
    """Main entry point."""
    import argparse

    parser = argparse.ArgumentParser(description='Build questionnaire monolith payload')
    parser.add_argument(
        '--output',
        '-o',
        default=None,
        help='Output path for the questionnaire monolith file'
    )

    args = parser.parse_args()

    forge = MonolithForge()
    success = forge.build(args.output)

    sys.exit(0 if success else 1)

if __name__ == '__main__':
    main()



================================================================================
FILE: scripts/operations/create_deployment_zip.py
================================================================================

#!/usr/bin/env python3
"""
Create a deployment zip file with only the required files for maximum performance.
Excludes deprecated files, documentation, tests, and development files.
"""

import os
import zipfile
from pathlib import Path

# Base directory
BASE_DIR = Path(__file__).parent.parent

# Essential documentation files to include (all others excluded)
ESSENTIAL_DOCS = {
    'README.md',
    'QUICKSTART.md',
}

```

```
# Files and directories to exclude
EXCLUDE_PATTERNS = {
    # Deprecated files
    'ORCHESTRATOR_MONILITH.py',
    'docs/README_MONOLITH.md',

    # Documentation files (not needed for deployment)
    '*.md',
    'DOCUMENTATION_OVERVIEW.txt',

    # Development and testing
    'tests/',
    'examples/',
    '.github/',
    '.augment/',

    # IDE and dev tools
    '.vscode/',
    '.DS_Store',
    '.gitignore',
    '.importlinter',
    '.pre-commit-config.yaml',
    '.python-version',

    # Build and cache files
    '__pycache__/',
    '*.pyc',
    '*.pyo',
    '*.pyd',
    '.pytest_cache/',
    '.mypy_cache/',
    '.coverage',
    'htmlcov/',

    # Environment files
    '.env',
    '.env.example',
    '.env.local',

    # Git directory
    '.git/',

    # Package management
    'minipdm/',  # Separate subproject, not needed for runtime

    # Development scripts
    'scripts/verify_dependencies.py',
    'scripts/setup.sh',
    'scripts/update_imports.py',
    'atroz_quickstart.sh',

    # Tools directory (development only)
    'tools/',
}

# Essential files and directories to include
```

```
INCLUDE_PATTERNS = {
    # Main source code
    'src/',

    # Compatibility shims (needed for backward compatibility)
    'orchestrator/',
    'concurrency/',
    'core/',
    'executors/',
    'contracts/',
    'validation/',
    'scoring/',

    # Configuration files
    'config/',

    # Data files
    'data/',

    # Essential scripts
    'scripts/create_deployment_zip.py',  # This script itself

    # Root level compatibility shims
    'aggregation.py',
    'contracts.py',
    'document_ingestion.py',
    'embedding_policy.py',
    'evidence_registry.py',
    'json_contract_loader.py',
    'macro_prompts.py',
    'meso_cluster_analysis.py',
    'micro_prompts.py',
    'policy_processor.py',
    'qmcm_hooks.py',
    'recommendation_engine.py',
    'runtime_error_fixes.py',
    'schema_validator.py',
    'seed_factory.py',
    'signature_validator.py',
    'validation_engine.py',

    # Package files
    'setup.py',
    'pyproject.toml',
    'requirements.txt',
    'requirements_atroz.txt',
    'constraints.txt',
    'Makefile',

    # Symlinks
    'rules',
    'schemas',

    # Essential documentation (minimal)
    'README.md',
    'QUICKSTART.md',
```

```
2241: }
2242:
2243: def should_include(path: Path, base: Path) -> bool:
2244:     """Determine if a file should be included in the deployment zip."""
2245:     relative_path = path.relative_to(base)
2246:     path_str = str(relative_path)
2247:
2248:     # Check if it's the deprecated ORCHESTRATOR_MONILITH
2249:     if 'ORCHESTRATOR_MONILITH.py' in path_str:
2250:         return False
2251:
2252:     # Exclude markdown files except essential ones
2253:     if path_str.endswith('.md'):
2254:         return path_str in ESSENTIAL_DOCS
2255:
2256:     # Check exclude patterns
2257:     for pattern in EXCLUDE_PATTERNS:
2258:         if pattern.endswith('/'):
2259:             # Directory pattern - match at start of path components
2260:             pattern_name = pattern.rstrip('/')
2261:             if path_str.startswith(pattern_name + '/') or path_str == pattern_name:
2262:                 return False
2263:         elif pattern.startswith('*.'):
2264:             # Extension pattern
2265:             if path_str.endswith(pattern[1:]):
2266:                 return False
2267:         elif '/' in pattern:
2268:             # Path pattern - exact match
2269:             if path_str == pattern or path_str.startswith(pattern + '/'):
2270:                 return False
2271:         else:
2272:             # Filename pattern - match exact filename component
2273:             parts = Path(path_str).parts
2274:             if pattern in parts:
2275:                 return False
2276:
2277:     # Check include patterns
2278:     for pattern in INCLUDE_PATTERNS:
2279:         if pattern.endswith('/'):
2280:             # Directory pattern
2281:             if path_str.startswith(pattern.rstrip('/')):
2282:                 return True
2283:         elif pattern == path_str:
2284:             # Exact match
2285:             return True
2286:
2287:     # If in src/ directory, include by default
2288:     return bool(path_str.startswith('src/'))
2289:
2290: def create_deployment_zip(output_path: Path) -> None:
2291:     """Create a deployment zip file."""
2292:     print(f"Creating deployment zip at {output_path}")
2293:
2294:     included_files: list[str] = []
2295:     excluded_files: list[str] = []
2296:
```

```
    with zipfile.ZipFile(output_path, 'w', zipfile.ZIP_DEFLATED) as zipf:
        for root, dirs, files in os.walk(BASE_DIR):
            root_path = Path(root)

            # Skip excluded directories
            dirs_to_remove = []
            for d in dirs:
                dir_path = root_path / d
                if not should_include(dir_path, BASE_DIR):
                    dirs_to_remove.append(d)

            for d in dirs_to_remove:
                dirs.remove(d)

            # Add files
            for file in files:
                file_path = root_path / file

                if should_include(file_path, BASE_DIR):
                    arcname = file_path.relative_to(BASE_DIR)
                    zipf.write(file_path, arcname)
                    included_files.append(str(arcname))
                else:
                    excluded_files.append(str(file_path.relative_to(BASE_DIR)))

    print("\nâ\234\205 Deployment zip created successfully!")
    print(f"   Output: {output_path}")
    print(f"   Size: {output_path.stat().st_size / 1024 / 1024:.2f} MB")
    print(f"   Included files: {len(included_files)}")
    print(f"   Excluded files: {len(excluded_files)}")

    # Print summary
    print("\nð\237\223¦ Included components:")
    components = set()
    for f in included_files:
        component = f.split('/')[0] if '/' in f else f
        components.add(component)
    for comp in sorted(components):
        count = sum(1 for f in included_files if f.startswith(comp))
        print(f"   - {comp}: {count} files")

    print("\nð\237\232« Excluded deprecated/development files:")
    print(f"   - Total excluded: {len(excluded_files)}")

    # Count different types of excluded files
    deprecated_count = sum(1 for f in excluded_files if 'MONOLITH' in f)
    doc_count = sum(1 for f in excluded_files if f.endswith('.md'))
    test_count = sum(1 for f in excluded_files if f.startswith('tests/'))
    example_count = sum(1 for f in excluded_files if f.startswith('examples/'))

    print(f"   - Deprecated files: {deprecated_count}")
    print(f"   - Documentation: {doc_count}")
    print(f"   - Tests: {test_count}")
    print(f"   - Examples: {example_count}")

    # Save manifest
```

```
    manifest_path = output_path.with_suffix('.txt')
    with open(manifest_path, 'w') as f:
        f.write("SAAAAAA Deployment Package - File Manifest\n")
        f.write("=" * 60 + "\n\n")
        f.write(f"Total files: {len(included_files)}\n\n")
        f.write("Included files:\n")
        f.write("-" * 60 + "\n")
        for file in sorted(included_files):
            f.write(f"{file}\n")

    print(f"\nð\237\223\204 Manifest saved to: {manifest_path}")

if __name__ == "__main__":
    output_zip = BASE_DIR / "farfan_pipeline-deployment.zip"
    create_deployment_zip(output_zip)
    print("\nâ\234¨ Deployment package ready for production!")



================================================================================
FILE: scripts/operations/create_graphs.py
================================================================================

import matplotlib.pyplot as plt
import networkx as nx

def create_atroz_graph(graph_data, filename, layout='spring'):
    """
    Generates and saves a graph with the Atroz dashboard aesthetic.
    """
    plt.style.use('dark_background')
    fig, ax = plt.subplots(figsize=(12, 8))
    fig.patch.set_facecolor('#0A0A0A')

    G = nx.DiGraph()
    for edge in graph_data['edges']:
        G.add_edge(edge[0], edge[1])

    if layout == 'spring':
        pos = nx.spring_layout(G, seed=42)
    elif layout == 'shell':
        pos = nx.shell_layout(G)
    elif layout == 'kamada_kawai':
        pos = nx.kamada_kawai_layout(G)
    else:
        pos = nx.spring_layout(G, seed=42)


    node_colors = ['#00D4FF' if node not in graph_data.get('error_nodes', []) else '#C41E3A' for node in G.nodes()]
    edge_colors = ['#B2642E' for _ in G.edges()]

    nx.draw_networkx_nodes(G, pos, node_color=node_colors, node_size=3000, ax=ax)
    nx.draw_networkx_edges(G, pos, edge_color=edge_colors, width=1.5, arrowsize=20, ax=ax)
    nx.draw_networkx_labels(G, pos, font_family='JetBrains Mono', font_size=10, font_color='#E5E7EB', ax=ax)

```

```
    ax.set_title(graph_data['title'], fontname='JetBrains Mono', fontsize=16, color='#E5E7EB')
    plt.savefig(filename, bbox_inches='tight', facecolor=fig.get_facecolor(), edgecolor='none')
    plt.close()

if __name__ == '__main__':
    control_flow_data = {
        'title': 'Control-Flow Graph',
        'edges': [
            ('pdf_path + config', 'run SPC ingestion'),
            ('run SPC ingestion', 'adapter â\206\222 PreprocessedDocument'),
            ('adapter â\206\222 PreprocessedDocument', 'validate chunk graph'),
            ('validate chunk graph', 'record ingestion metadata'),
            ('validate chunk graph', 'Abort run'),
            ('record ingestion metadata', 'emit PreprocessedDocument')
        ],
        'error_nodes': ['Abort run']
    }
    create_atroz_graph(control_flow_data, 'docs/phases/phase_1/images/control_flow.png')

    data_flow_data = {
        'title': 'Data-Flow Graph',
        'edges': [
            ('pdf_path', 'SPCIngestion'),
            ('config', 'SPCIngestion'),
            ('SPCIngestion', 'Adapter'),
            ('Adapter', 'PreprocessedDocument'),
            ('PreprocessedDocument', 'Validator'),
            ('Validator', 'OrchestratorContext'),
            ('Validator', 'VerificationManifest')
        ]
    }
    create_atroz_graph(data_flow_data, 'docs/phases/phase_1/images/data_flow.png', layout='kamada_kawai')

    state_transition_data = {
        'title': 'State-Transition Graph',
        'edges': [
            ('Idle', 'Ingesting'),
            ('Ingesting', 'Adapting'),
            ('Ingesting', 'Faulted'),
            ('Adapting', 'Validating'),
            ('Adapting', 'Faulted'),
            ('Validating', 'Recording'),
            ('Validating', 'Faulted'),
            ('Recording', 'Emitting'),
            ('Emitting', 'Idle')
        ],
        'error_nodes': ['Faulted']
    }
    create_atroz_graph(state_transition_data, 'docs/phases/phase_1/images/state_transition.png', layout='shell')

    contract_linkage_data = {
        'title': 'Contract-Linkage Graph',
        'edges': [
            ('SPC-PIPELINE-V1', 'SPCIngestion'),
            ('CPP-ADAPTER-V1', 'Adapter'),
            ('SPC-CHUNK-V1', 'Validator'),
```

```
            ('PREPROC-V1', 'Validator'),
            ('VERIF-MANIFEST-V1', 'Recording')
        ]
    }
    create_atroz_graph(contract_linkage_data, 'docs/phases/phase_1/images/contract_linkage.png', layout='kamada_kawai')


================================================================================
FILE: scripts/operations/generate_all_v3_contracts.py
================================================================================

#!/usr/bin/env python3
"""
Generate all 300 v3 executor contracts (D1-Q1 through D6-Q5)

Uses D1-Q1.v3.FINAL.json as template, customizes for each question.
"""

import json
from pathlib import Path
from datetime import datetime

PROJECT_ROOT = Path(__file__).parent.parent

# Load template
template_path = PROJECT_ROOT / "config" / "executor_contracts" / "D1-Q1.v3.FINAL.json"
with open(template_path) as f:
    template = json.load(f)

# Load methods mapping
methods_mapping_path = PROJECT_ROOT / "executor_methods_mapping.json"
with open(methods_mapping_path) as f:
    methods_mapping = json.load(f)

# Load questionnaire monolith
monolith_path = PROJECT_ROOT / "data" / "questionnaire_monolith.json"
with open(monolith_path) as f:
    monolith = json.load(f)

micro_questions = monolith["blocks"]["micro_questions"]

# Define method roles (generic mapping by method name patterns)
def get_method_role(method_name):
    """Generate semantic role for a method based on its name."""
    role_map = {
        "diagnose": "diagnosis",
        "analyze": "analysis",
        "extract": "extraction",
        "parse": "parsing",
        "process": "processing",
        "match": "matching",
        "detect": "detection",
        "validate": "validation",
        "audit": "auditing",
        "evaluate": "evaluation",
```

```
        "compare": "comparison",
        "calculate": "calculation",
        "infer": "inference",
        "construct": "construction",
        "identify": "identification",
        "generate": "generation",
        "chunk": "chunking",
        "embed": "embedding",
        "classify": "classification",
        "trace": "tracing",
        "verify": "verification",
    }

    for keyword, role in role_map.items():
        if keyword in method_name.lower():
            return f"{method_name}_{role}"

    return method_name.replace("_", "_")

# Define provides mapping (group by class namespace)
def get_provides_key(class_name, method_name):
    """Generate dot-notation provides key for a method."""
    namespace_map = {
        "TextMiningEngine": "text_mining",
        "IndustrialPolicyProcessor": "industrial_policy",
        "CausalExtractor": "causal_extraction",
        "FinancialAuditor": "financial_audit",
        "PDETMunicipalPlanAnalyzer": "pdet_analysis",
        "PolicyContradictionDetector": "contradiction_detection",
        "BayesianNumericalAnalyzer": "bayesian_analysis",
        "SemanticProcessor": "semantic_processing",
        "OperationalizationAuditor": "operationalization",
        "BayesianMechanismInference": "bayesian_mechanism",
        "BayesianCounterfactualAuditor": "bayesian_counterfactual",
        "PDFProcessor": "pdf_processing",
        "TeoriaCambio": "teoria_cambio",
        "BeachEvidentialTest": "beach_test",
        "AdvancedDAGValidator": "dag_validation",
        "BayesFactorTable": "bayes_factor",
        "HierarchicalGenerativeModel": "hierarchical_model",
        "IndustrialGradeValidator": "industrial_validator",
        "PerformanceAnalyzer": "performance",
        "SemanticAnalyzer": "semantic_analysis",
        "AdaptivePriorCalculator": "adaptive_prior",
        "PolicyAnalysisEmbedder": "policy_embedder",
        "ReportingEngine": "reporting",
        "TemporalLogicVerifier": "temporal_logic",
        "CausalInferenceSetup": "causal_setup",
        "MechanismPartExtractor": "mechanism_extraction",
        "CDAFFramework": "cdaf_framework",
        "PolicyTextProcessor": "text_processing",
        "ConfigLoader": "config_management",
    }

    namespace = namespace_map.get(class_name, class_name.lower())
    method_key = method_name.lstrip("_").replace("__", "_")
```

```
2577:
2578:     return f"{namespace}.{method_key}"
2579:
2580: # Generate contracts
2581: output_dir = PROJECT_ROOT / "config" / "executor_contracts"
2582: output_dir.mkdir(parents=True, exist_ok=True)
2583:
2584: generated_count = 0
2585: errors = []
2586:
2587: for question in micro_questions:
2588:     base_slot = question["base_slot"]
2589:     question_id = question["question_id"]
2590:
2591:     try:
2592:         # Get methods for this question
2593:         if base_slot not in methods_mapping:
2594:             errors.append(f"{base_slot}: No methods found in mapping")
2595:             continue
2596:
2597:         question_methods = methods_mapping[base_slot]
2598:         method_count = len(question_methods)
2599:
2600:         # Create contract from template
2601:         contract = json.loads(json.dumps(template))  # Deep copy
2602:
2603:         # Update identity
2604:         contract["identity"]["base_slot"] = base_slot
2605:         contract["identity"]["question_id"] = question_id
2606:         contract["identity"]["dimension_id"] = question.get("identity", {}).get("dimension_id", "DIM01")
2607:         contract["identity"]["policy_area_id"] = question.get("policy_area_id", "PA01")
2608:         contract["identity"]["created_at"] = datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ")
2609:
2610:         # Update executor_binding
2611:         executor_class = base_slot.replace("-", "_") + "_Executor"
2612:         contract["executor_binding"]["executor_class"] = executor_class
2613:
2614:         # Build methods array
2615:         methods_array = []
2616:         for i, method_info in enumerate(question_methods, start=1):
2617:             class_name = method_info["class"]
2618:             method_name = method_info["method"]
2619:
2620:             methods_array.append({
2621:                 "class_name": class_name,
2622:                 "method_name": method_name,
2623:                 "priority": i,
2624:                 "provides": get_provides_key(class_name, method_name),
2625:                 "role": get_method_role(method_name)
2626:             })
2627:
2628:         # Update method_binding
2629:         contract["method_binding"] = {
2630:             "orchestration_mode": "multi_method_pipeline",
2631:             "method_count": method_count,
2632:             "methods": methods_array,
```

```
            "note": f"All {method_count} methods extracted from {executor_class} in executors.py"
        }

        # Update question_context from monolith
        contract["question_context"]["question_text"] = question.get("question_text", "")
        contract["question_context"]["question_type"] = question.get("question_type", "micro")
        contract["question_context"]["scoring_modality"] = question.get("scoring_modality", "TYPE_A")
        contract["question_context"]["expected_output_type"] = question.get("expected_output_type", "score")
        contract["question_context"]["patterns"] = question.get("patterns", [])
        contract["question_context"]["expected_elements"] = question.get("expected_elements", [])
        contract["question_context"]["validations"] = question.get("validations", {})

        # Update output_contract schema constants
        contract["output_contract"]["schema"]["properties"]["base_slot"]["const"] = base_slot
        contract["output_contract"]["schema"]["properties"]["question_id"]["const"] = question_id
        contract["output_contract"]["schema"]["properties"]["question_global"]["const"] = question.get("question_global")
        contract["output_contract"]["schema"]["properties"]["policy_area_id"]["const"] = question.get("policy_area_id")
        contract["output_contract"]["schema"]["properties"]["dimension_id"]["const"] = question.get("identity", {}).get("dimension_id")
        contract["output_contract"]["schema"]["properties"]["cluster_id"]["const"] = question.get("identity", {}).get("cluster_id")

        # Update methodological_depth methods
        methodological_methods = []
        for i, method_info in enumerate(question_methods, start=1):
            class_name = method_info["class"]
            method_name = method_info["method"]

            # Find corresponding method from template (if exists) or use generic structure
            template_method = None
            for tm in template["output_contract"]["human_readable_output"]["methodological_depth"]["methods"]:
                if tm["class_name"] == class_name and tm["method_name"] == method_name:
                    template_method = tm
                    break

            if template_method:
                # Use existing documentation
                methodological_methods.append(template_method)
            else:
                # Create generic documentation
                methodological_methods.append({
                    "method_name": method_name,
                    "class_name": class_name,
                    "priority": i,
                    "role": get_method_role(method_name),
                    "epistemological_foundation": {
                        "paradigm": f"{class_name} analytical paradigm",
                        "ontological_basis": f"Analysis via {class_name}.{method_name}",
                        "epistemological_stance": "Empirical-analytical approach",
                        "theoretical_framework": [
                            f"Method {method_name} implements structured analysis for {base_slot}"
                        ],
                        "justification": f"This method contributes to {base_slot} analysis"
                    },
                    "technical_approach": {
                        "method_type": "analytical_processing",
                        "algorithm": f"{class_name}.{method_name} algorithm",
                        "steps": [
```

```
                            {"step": 1, "description": f"Execute {method_name}"},
                            {"step": 2, "description": "Process results"},
                            {"step": 3, "description": "Return structured output"}
                        ],
                        "assumptions": [
                            "Input data is preprocessed and valid"
                        ],
                        "limitations": [
                            "Method-specific limitations apply"
                        ],
                        "complexity": "O(n) where n=input size"
                    },
                    "output_interpretation": {
                        "output_structure": {
                            "result": f"Structured output from {method_name}"
                        },
                        "interpretation_guide": {
                            "high_confidence": "â\211¥0.8: Strong evidence",
                            "medium_confidence": "0.5-0.79: Moderate evidence",
                            "low_confidence": "<0.5: Weak evidence"
                        },
                        "actionable_insights": [
                            f"Use {method_name} results for downstream analysis"
                        ]
                    }
                })

        contract["output_contract"]["human_readable_output"]["methodological_depth"]["methods"] = methodological_methods

        # Update human_answer_structure metadata
        contract["human_answer_structure"]["evidence_structure_schema"]["properties"]["metadata"]["properties"]["methods_executed"]["const"] = method_count
        contract["human_answer_structure"]["concrete_example"]["metadata"]["methods_executed"] = method_count

        # Update traceability
        contract["traceability"]["json_path"] = f"blocks.micro_questions[{question.get('question_global', 0) - 1}]"
        contract["traceability"]["method_source"] = f"src/farfan_pipeline/core/orchestrator/executors.py:{executor_class}"

        # Write contract
        output_path = output_dir / f"{base_slot}.v3.json"
        with open(output_path, 'w') as f:
            json.dump(contract, f, indent=2, ensure_ascii=False)

        generated_count += 1
        print(f"â\234\205 {generated_count:3d}/300 {base_slot:8s} ({method_count:2d} methods) â\206\222 {output_path.name}")

    except Exception as e:
        errors.append(f"{base_slot}: {str(e)}")
        print(f"â\235\214 {base_slot}: Error - {str(e)}")

# Summary
print("\n" + "="*80)
print(f"â\234\205 Generated: {generated_count}/300 contracts")
if errors:
    print(f"â\235\214 Errors: {len(errors)}")
    for error in errors[:10]:  # Show first 10 errors
        print(f"   - {error}")
```

```
2745: else:
2746:     print("ð\237\216\211 All contracts generated successfully!")
2747: print("="*80)
2748:
2749: # Save generation log
2750: log_path = PROJECT_ROOT / "docs" / "contract_generation_log.json"
2751: with open(log_path, 'w') as f:
2752:     json.dump({
2753:         "timestamp": datetime.utcnow().isoformat(),
2754:         "generated_count": generated_count,
2755:         "total_expected": 300,
2756:         "errors": errors,
2757:         "template_used": str(template_path),
2758:         "output_directory": str(output_dir)
2759:     }, f, indent=2)
2760:
2761: print(f"\nð\237\223\204 Generation log saved: {log_path}")
2762:
2763:
2764:
2765: ================================================================================
2766: FILE: scripts/operations/generate_d1q1_final.py
2767: ================================================================================
2768:
2769: #!/usr/bin/env python3
2770: """
2771: Generate the final D1-Q1.v3.CANONICAL.json with:
2772: 1. Correct method_binding.methods array (17 methods)
2773: 2. Complete human_answer_structure section
2774: """
2775:
2776: import json
2777: from pathlib import Path
2778:
2779: PROJECT_ROOT = Path(__file__).parent.parent
2780:
2781: # Load current contract
2782: current_contract_path = PROJECT_ROOT / "config" / "executor_contracts" / "D1-Q1.v3.CANONICAL.json"
2783: with open(current_contract_path) as f:
2784:     contract = json.load(f)
2785:
2786: # Load methods mapping
2787: methods_mapping_path = PROJECT_ROOT / "executor_methods_mapping.json"
2788: with open(methods_mapping_path) as f:
2789:     methods_mapping = json.load(f)
2790:
2791: d1q1_methods = methods_mapping["D1-Q1"]
2792:
2793: # Define role mapping for each method
2794: method_roles = {
2795:     "diagnose_critical_links": "critical_link_diagnosis",
2796:     "_analyze_link_text": "link_context_analysis",
2797:     "process": "industrial_policy_pattern_processing",
2798:     "_match_patterns_in_sentences": "sentence_level_pattern_matching",
2799:     "_extract_point_evidence": "point_evidence_extraction",
2800:     "_extract_goals": "goal_extraction",
```

```
    "_parse_goal_context": "goal_contextualization",
    "_parse_amount": "financial_amount_parsing",
    "_extract_financial_amounts": "pdet_specific_financial_extraction",
    "_extract_from_budget_table": "structured_budget_table_extraction",
    "_extract_quantitative_claims": "quantitative_claim_extraction",
    "_parse_number": "numeric_value_parsing",
    "_statistical_significance_test": "statistical_significance_testing",
    "evaluate_policy_metric": "bayesian_metric_evaluation",
    "compare_policies": "bayesian_policy_comparison",
    "chunk_text": "text_chunking_for_embedding",
    "embed_single": "semantic_embedding",
}

# Define provides mapping (dot-notation keys)
method_provides = {
    "diagnose_critical_links": "text_mining.critical_links",
    "_analyze_link_text": "text_mining.link_analysis",
    "process": "industrial_policy.structure",
    "_match_patterns_in_sentences": "industrial_policy.sentence_patterns",
    "_extract_point_evidence": "industrial_policy.processed_evidence",
    "_extract_goals": "causal_extraction.goals",
    "_parse_goal_context": "causal_extraction.goal_contexts",
    "_parse_amount": "financial_audit.amounts",
    "_extract_financial_amounts": "pdet_analysis.financial_data",
    "_extract_from_budget_table": "pdet_analysis.budget_tables",
    "_extract_quantitative_claims": "contradiction_detection.quantitative_claims",
    "_parse_number": "contradiction_detection.parsed_numbers",
    "_statistical_significance_test": "contradiction_detection.significance_tests",
    "evaluate_policy_metric": "bayesian_analysis.policy_metrics",
    "compare_policies": "bayesian_analysis.comparisons",
    "chunk_text": "semantic_processing.chunks",
    "embed_single": "semantic_processing.embeddings",
}

# Build methods array
methods_array = []
for i, method_info in enumerate(d1q1_methods, start=1):
    class_name = method_info["class"]
    method_name = method_info["method"]

    methods_array.append({
        "class_name": class_name,
        "method_name": method_name,
        "priority": i,
        "provides": method_provides.get(method_name, f"{class_name}.{method_name}"),
        "role": method_roles.get(method_name, f"{class_name}_{method_name}")
    })

# Update method_binding
contract["method_binding"] = {
    "orchestration_mode": "multi_method_pipeline",
    "method_count": 17,
    "methods": methods_array,
    "note": "All 17 methods extracted from D1_Q1_QuantitativeBaselineExtractor in executors.py"
}

```

```
# Add human_answer_structure
contract["human_answer_structure"] = {
    "description": "Expected structure of evidence dict after all 17 methods execute and evidence is assembled according to assembly_rules",
    "assembly_flow": {
        "step_1_method_execution": "17 methods execute in priority order, outputs stored with dot-notation keys",
        "step_2_evidence_assembly": "EvidenceAssembler merges outputs according to assembly_rules",
        "step_3_validation": "EvidenceValidator checks against validation_rules",
        "step_4_output_generation": "Phase2QuestionResult constructed with evidence, validation, trace"
    },
    "evidence_structure_schema": {
        "type": "object",
        "description": "Assembled evidence after all methods complete",
        "properties": {
            "elements_found": {
                "type": "array",
                "description": "Concatenated evidence elements from multiple methods (assembly_rules target)",
                "items": {
                    "type": "object",
                    "properties": {
                        "element_id": {"type": "string", "example": "E-001"},
                        "type": {
                            "type": "string",
                            "enum": [
                                "fuentes_oficiales",
                                "indicadores_cuantitativos",
                                "series_temporales_aÃ±os",
                                "cobertura_territorial_especificada",
                                "financial_amounts",
                                "policy_goals",
                                "causal_links"
                            ]
                        },
                        "value": {"type": "string", "example": "DANE"},
                        "confidence": {"type": "number", "minimum": 0, "maximum": 1},
                        "source_method": {"type": "string", "example": "IndustrialPolicyProcessor._extract_point_evidence"},
                        "sentence_id": {"type": "integer"},
                        "context": {"type": "string"}
                    }
                },
                "example_count": "Expected 15-50 elements for a complete diagnostic"
            },
            "elements_summary": {
                "type": "object",
                "properties": {
                    "total_count": {"type": "integer"},
                    "by_type": {
                        "type": "object",
                        "properties": {
                            "fuentes_oficiales": {"type": "integer", "minimum_expected": 2},
                            "indicadores_cuantitativos": {"type": "integer", "minimum_expected": 3},
                            "series_temporales_aÃ±os": {"type": "integer", "minimum_expected": 3},
                            "cobertura_territorial_especificada": {"type": "integer", "minimum_expected": 1}
                        }
                    }
                }
            },
```

```
            "confidence_scores": {
                "type": "object",
                "description": "Aggregated confidence metrics (weighted_mean strategy)",
                "properties": {
                    "mean": {"type": "number"},
                    "std": {"type": "number"},
                    "min": {"type": "number"},
                    "max": {"type": "number"},
                    "by_method": {
                        "type": "object",
                        "description": "Average confidence per analyzer class"
                    }
                }
            },
            "pattern_matches": {
                "type": "array",
                "description": "Aggregated pattern matches from text mining methods",
                "items": {
                    "type": "object",
                    "properties": {
                        "pattern_id": {"type": "string"},
                        "count": {"type": "integer"},
                        "avg_confidence": {"type": "number"}
                    }
                }
            },
            "critical_links": {
                "type": "array",
                "description": "Causal links extracted by TextMiningEngine",
                "items": {
                    "type": "object",
                    "properties": {
                        "cause": {"type": "string"},
                        "effect": {"type": "string"},
                        "criticality": {"type": "number"},
                        "coherence": {"type": "number"}
                    }
                }
            },
            "financial_summary": {
                "type": "object",
                "description": "Aggregated financial data from FinancialAuditor and PDETMunicipalPlanAnalyzer",
                "properties": {
                    "total_budget_cop": {"type": "number"},
                    "amounts_found": {"type": "integer"},
                    "by_category": {
                        "type": "object",
                        "properties": {
                            "SGR": {"type": "number"},
                            "recursos_propios": {"type": "number"},
                            "transferencias": {"type": "number"}
                        }
                    }
                }
            },
            "goals_summary": {
```

```
                "type": "object",
                "description": "Policy goals extracted by CausalExtractor",
                "properties": {
                    "total_goals": {"type": "integer"},
                    "quantified_goals": {"type": "integer"},
                    "goals_with_complete_context": {"type": "integer"}
                }
            },
            "contradictions": {
                "type": "object",
                "description": "Results from PolicyContradictionDetector",
                "properties": {
                    "found": {"type": "integer"},
                    "tests_performed": {"type": "integer"},
                    "interpretation": {"type": "string"}
                }
            },
            "bayesian_insights": {
                "type": "object",
                "description": "Results from BayesianNumericalAnalyzer",
                "properties": {
                    "metrics_with_high_uncertainty": {"type": "array"},
                    "significant_comparisons": {"type": "integer"}
                }
            },
            "semantic_processing": {
                "type": "object",
                "description": "Results from SemanticProcessor",
                "properties": {
                    "chunks_created": {"type": "integer"},
                    "embeddings_generated": {"type": "integer"},
                    "avg_semantic_similarity_to_query": {"type": "number"}
                }
            },
            "metadata": {
                "type": "object",
                "properties": {
                    "methods_executed": {"type": "integer", "const": 17},
                    "execution_time_ms": {"type": "number"},
                    "document_length": {"type": "integer"},
                    "analysis_timestamp": {"type": "string", "format": "date-time"}
                }
            }
        }
    },
    "concrete_example": {
        "elements_found": [
            {
                "element_id": "E-001",
                "type": "fuentes_oficiales",
                "value": "DANE",
                "confidence": 0.95,
                "source_method": "IndustrialPolicyProcessor._extract_point_evidence",
                "source_sentence": "segÃºn datos de DANE para el aÃ±o 2022",
                "sentence_id": 45,
                "position": {"start": 123, "end": 145}
```

```
            },
            {
                "element_id": "E-002",
                "type": "indicadores_cuantitativos",
                "value": "tasa de VBG: 12.3%",
                "normalized_value": 12.3,
                "unit": "%",
                "confidence": 0.89,
                "source_method": "PolicyContradictionDetector._extract_quantitative_claims",
                "bayesian_posterior": {
                    "mean": 0.123,
                    "ci_95": [0.11, 0.145]
                },
                "sentence_id": 45
            },
            {
                "element_id": "E-003",
                "type": "series_temporales_aÃ±os",
                "years": [2020, 2021, 2022],
                "confidence": 0.92,
                "source_method": "TextMiningEngine.diagnose_critical_links"
            },
            {
                "element_id": "E-004",
                "type": "cobertura_territorial_especificada",
                "coverage": "municipal - zona rural y urbana",
                "confidence": 0.88,
                "source_method": "CausalExtractor._parse_goal_context"
            }
        ],
        "elements_summary": {
            "total_count": 38,
            "by_type": {
                "fuentes_oficiales": 5,
                "indicadores_cuantitativos": 12,
                "series_temporales_aÃ±os": 4,
                "cobertura_territorial_especificada": 1,
                "financial_amounts": 8,
                "policy_goals": 7,
                "causal_links": 5
            }
        },
        "confidence_scores": {
            "mean": 0.876,
            "std": 0.089,
            "min": 0.72,
            "max": 0.98,
            "by_method": {
                "TextMiningEngine": 0.83,
                "IndustrialPolicyProcessor": 0.91,
                "CausalExtractor": 0.79,
                "FinancialAuditor": 0.94,
                "PDETMunicipalPlanAnalyzer": 0.88,
                "PolicyContradictionDetector": 0.90,
                "BayesianNumericalAnalyzer": 0.92,
                "SemanticProcessor": 0.85
```

```
            }
        },
        "pattern_matches": [
            {"pattern_id": "PAT-Q001-000", "count": 3, "avg_confidence": 0.87},
            {"pattern_id": "PAT-Q001-002", "count": 5, "avg_confidence": 0.95}
        ],
        "critical_links": [
            {
                "cause": "alta tasa de VBG",
                "effect": "baja autonomÃ-a econÃ³mica",
                "criticality": 0.87,
                "coherence": 0.82
            }
        ],
        "financial_summary": {
            "total_budget_cop": 850000000.0,
            "amounts_found": 12,
            "by_category": {
                "SGR": 250000000.0,
                "recursos_propios": 180000000.0
            }
        },
        "goals_summary": {
            "total_goals": 7,
            "quantified_goals": 5,
            "goals_with_complete_context": 4
        },
        "contradictions": {
            "found": 0,
            "tests_performed": 15,
            "interpretation": "No statistical contradictions in quantitative claims"
        },
        "bayesian_insights": {
            "metrics_with_high_uncertainty": [],
            "significant_comparisons": 1
        },
        "semantic_processing": {
            "chunks_created": 45,
            "embeddings_generated": 45,
            "avg_semantic_similarity_to_query": 0.78
        },
        "metadata": {
            "methods_executed": 17,
            "execution_time_ms": 2845,
            "document_length": 15230,
            "analysis_timestamp": "2025-11-26T12:34:56Z"
        }
    },
    "validation_against_expected_elements": {
        "cobertura_territorial_especificada": {
            "required": True,
            "found_in_example": True,
            "example_element_id": "E-004"
        },
        "fuentes_oficiales": {
            "minimum": 2,
```

```
            "found_in_example": 5,
            "status": "PASS"
        },
        "indicadores_cuantitativos": {
            "minimum": 3,
            "found_in_example": 12,
            "status": "PASS"
        },
        "series_temporales_aÃ±os": {
            "minimum": 3,
            "found_in_example": 4,
            "status": "PASS"
        },
        "overall_validation_result": "PASS - All required and minimum elements present"
    },
    "template_variable_bindings": {
        "description": "These variables are available for human_readable_output template",
        "variables": {
            "{evidence.elements_found_count}": 38,
            "{score}": "Calculated by scorer based on elements",
            "{quality_level}": "ALTO",
            "{evidence.confidence_scores.mean}": "87.6%",
            "{evidence.pattern_matches_count}": 14,
            "{evidence.official_sources_count}": 5,
            "{evidence.quantitative_indicators_count}": 12,
            "{evidence.temporal_series_count}": 4,
            "{evidence.territorial_coverage}": "municipal - zona rural y urbana"
        }
    },
    "usage_notes": {
        "for_developers": "This structure shows the expected evidence dict after BaseExecutorWithContract._execute_v3() completes all 17 method executions a
nd evidence assembly.",
        "for_validators": "Use this to verify that actual execution output matches expected structure.",
        "for_auditors": "This provides traceability from raw method outputs to final assembled evidence."
    }
}

# Update traceability
contract["traceability"]["contract_generation_method"] = "canonical_prompt_v3_with_multi_method_refactoring"
contract["traceability"]["provenance_note"] = "This contract was generated with full multi-method orchestration support. The method_binding.methods array co
ntains all 17 methods from D1_Q1_QuantitativeBaselineExtractor, and human_answer_structure documents the expected evidence output after execution."

# Write updated contract
output_path = PROJECT_ROOT / "config" / "executor_contracts" / "D1-Q1.v3.FINAL.json"
with open(output_path, 'w') as f:
    json.dump(contract, f, indent=2, ensure_ascii=False)

print(f"â\234\205 Generated: {output_path}")
print(f"   - method_binding.methods: {len(methods_array)} methods")
print(f"   - human_answer_structure: Complete with schema, example, and validation")
print(f"   - File size: {output_path.stat().st_size:,} bytes")



================================================================================
FILE: scripts/operations/generate_dependency_files.py
```

```
3191: ================================================================================
3192:
3193: #!/usr/bin/env python3
3194: """
3195: Generate comprehensive dependency files for SAAAAAA project.
3196:
3197: Generates:
3198: - requirements-core.txt: Core runtime dependencies with exact pins
3199: - requirements-dev.txt: Development and testing dependencies
3200: - requirements-optional.txt: Optional runtime dependencies
3201: - requirements-constraints.txt: Full constraint file with all transitive deps
3202: - Updated pyproject.toml with proper dependency groups
3203: """
3204:
3205: import json
3206: import sys
3207: from pathlib import Path
3208: from typing import Dict, List, Set
3209:
3210:
3211: def load_audit_report(project_root: Path) -> Dict:
3212:     """Load the dependency audit report."""
3213:     report_file = project_root / "dependency_audit_report.json"
3214:     if not report_file.exists():
3215:         print("Error: dependency_audit_report.json not found. Run audit_dependencies.py first.")
3216:         sys.exit(1)
3217:
3218:     with open(report_file, 'r') as f:
3219:         return json.load(f)
3220:
3221:
3222: def get_current_versions(requirements_file: Path) -> Dict[str, str]:
3223:     """Extract package versions from existing requirements.txt."""
3224:     versions = {}
3225:
3226:     if not requirements_file.exists():
3227:         return versions
3228:
3229:     with open(requirements_file, 'r') as f:
3230:         for line in f:
3231:             line = line.strip()
3232:             if not line or line.startswith('#'):
3233:                 continue
3234:
3235:             if '==' in line:
3236:                 pkg, ver = line.split('==', 1)
3237:                 versions[pkg.lower().strip()] = ver.strip()
3238:
3239:     return versions
3240:
3241:
3242: # Core runtime dependencies with known compatibility for Python 3.10-3.12
3243: CORE_DEPENDENCIES = {
3244:     # Web frameworks
3245:     "flask": "3.0.3",
3246:     "fastapi": "0.115.6",
```

```
    "uvicorn": "0.34.0",
    "httpx": "0.28.1",
    "sse-starlette": "2.2.1",
    "werkzeug": "3.0.6",

    # Configuration
    "pyyaml": "6.0.2",
    "python-dotenv": "1.0.1",
    "typer": "0.15.1",

    # Data processing - Core
    "numpy": "2.2.1",
    "scipy": "1.15.1",
    "pandas": "2.2.3",
    "polars": "1.19.0",
    "pyarrow": "19.0.0",

    # Machine Learning - note: tensorflow and torch need special handling for Python 3.12
    "scikit-learn": "1.6.1",
    # tensorflow is omitted - needs Python <3.12 or version >=2.16
    # torch is omitted - needs to be installed separately based on platform

    # NLP
    "transformers": "4.48.3",
    "sentence-transformers": "3.3.1",
    "spacy": "3.8.3",

    # Graph Analysis
    "networkx": "3.4.2",

    # Bayesian Analysis - note: pymc has strict version requirements
    # pymc omitted for now - complex dependencies

    # PDF Processing
    "pdfplumber": "0.11.4",
    "PyPDF2": "3.0.1",
    "PyMuPDF": "1.25.2",
    "python-docx": "1.1.2",

    # Data Validation
    "jsonschema": "4.23.0",
    "pydantic": "2.10.6",

    # Monitoring & Logging
    "structlog": "24.4.0",
    "tenacity": "9.0.0",

    # Security & Hashing
    "blake3": "0.4.1",

    # Type hints
    "typing-extensions": "4.12.2",
}

OPTIONAL_DEPENDENCIES = {
    # WebSocket Support
```

```
    "flask-cors": "6.0.0",
    "flask-socketio": "5.4.1",
    "python-socketio": "5.14.1",
    "gevent": "24.11.1",
    "gevent-websocket": "0.10.1",

    # Authentication
    "pyjwt": "2.10.1",

    # Advanced graph analysis
    "igraph": "0.11.8",
    "python-louvain": "0.16",
    "pydot": "3.0.4",

    # Causal inference - complex dependencies
    # "dowhy": "0.11.1",
    # "econml": "0.15.1",

    # Additional PDF
    "tabula-py": "2.10.0",
    "camelot-py": "0.11.0",

    # NLP Additional
    "nltk": "3.9.1",
    "sentencepiece": "0.2.0",
    "tiktoken": "0.8.0",
    "fuzzywuzzy": "0.18.0",
    "python-Levenshtein": "0.26.1",
    "langdetect": "1.0.9",

    # Database
    "redis": "5.2.1",
    "sqlalchemy": "2.0.37",

    # Production Server
    "gunicorn": "23.0.0",

    # Monitoring Advanced
    "prometheus-client": "0.21.1",
    "psutil": "6.1.1",
    "opentelemetry-api": "1.29.0",
    "opentelemetry-sdk": "1.29.0",
    # Note: opentelemetry-instrumentation-fastapi is beta - use with caution
    # Consider moving to dev/test if stability is an issue
    "opentelemetry-instrumentation-fastapi": "0.50b0",

    # HTML parsing
    "beautifulsoup4": "4.12.3",
}

DEV_DEPENDENCIES = {
    # Testing
    "pytest": "8.3.4",
    "pytest-cov": "6.0.0",
    "pytest-asyncio": "0.25.2",
    "hypothesis": "6.124.3",
```

```
    "schemathesis": "3.38.4",

    # Code Quality
    "black": "24.10.0",
    "ruff": "0.9.1",
    "flake8": "7.1.1",
    "mypy": "1.14.1",
    "pyright": "1.1.395",

    # Security
    "bandit": "1.8.0",

    # Tools
    "import-linter": "2.2",
}

DOCS_DEPENDENCIES = {
    "sphinx": "8.1.3",
    "sphinx-rtd-theme": "3.0.2",
    "myst-parser": "4.0.0",
}


def generate_core_requirements(output_file: Path, versions: Dict[str, str]):
    """Generate requirements-core.txt with exact pins."""
    print(f"Generating {output_file}...")

    with open(output_file, 'w') as f:
        f.write("# Core Runtime Dependencies - Exact Pins\n")
        f.write("# Generated by scripts/generate_dependency_files.py\n")
        f.write("# DO NOT EDIT MANUALLY - Update versions in the generator script\n\n")
        f.write("# This file contains ONLY critical runtime dependencies\n")
        f.write("# with exact version pins for reproducibility\n\n")

        for pkg, version in sorted(CORE_DEPENDENCIES.items()):
            f.write(f"{pkg}=={version}\n")

        f.write("\n# Notes:\n")
        f.write("# - tensorflow requires Python <3.12 or version >=2.16\n")
        f.write("# - torch should be installed separately based on platform\n")
        f.write("# - pymc has complex dependencies - install separately if needed\n")


def generate_optional_requirements(output_file: Path, versions: Dict[str, str]):
    """Generate requirements-optional.txt."""
    print(f"Generating {output_file}...")

    with open(output_file, 'w') as f:
        f.write("# Optional Runtime Dependencies - Exact Pins\n")
        f.write("# Generated by scripts/generate_dependency_files.py\n")
        f.write("# Install with: pip install -r requirements-optional.txt\n\n")

        for pkg, version in sorted(OPTIONAL_DEPENDENCIES.items()):
            f.write(f"{pkg}=={version}\n")


```

```
3415: def generate_dev_requirements(output_file: Path, versions: Dict[str, str]):
3416:     """Generate requirements-dev.txt."""
3417:     print(f"Generating {output_file}...")
3418:
3419:     with open(output_file, 'w') as f:
3420:         f.write("# Development & Testing Dependencies - Exact Pins\n")
3421:         f.write("# Generated by scripts/generate_dependency_files.py\n")
3422:         f.write("# Install with: pip install -r requirements-dev.txt\n\n")
3423:         f.write("# Include core dependencies\n")
3424:         f.write("-r requirements-core.txt\n\n")
3425:
3426:         for pkg, version in sorted(DEV_DEPENDENCIES.items()):
3427:             f.write(f"{pkg}=={version}\n")
3428:
3429:
3430: def generate_docs_requirements(output_file: Path, versions: Dict[str, str]):
3431:     """Generate requirements-docs.txt."""
3432:     print(f"Generating {output_file}...")
3433:
3434:     with open(output_file, 'w') as f:
3435:         f.write("# Documentation Dependencies - Exact Pins\n")
3436:         f.write("# Generated by scripts/generate_dependency_files.py\n")
3437:         f.write("# Install with: pip install -r requirements-docs.txt\n\n")
3438:
3439:         for pkg, version in sorted(DOCS_DEPENDENCIES.items()):
3440:             f.write(f"{pkg}=={version}\n")
3441:
3442:
3443: def generate_all_requirements(output_file: Path, versions: Dict[str, str]):
3444:     """Generate requirements-all.txt combining all dependencies."""
3445:     print(f"Generating {output_file}...")
3446:
3447:     with open(output_file, 'w') as f:
3448:         f.write("# All Dependencies - For Complete Installation\n")
3449:         f.write("# Generated by scripts/generate_dependency_files.py\n")
3450:         f.write("# Install with: pip install -r requirements-all.txt\n\n")
3451:
3452:         f.write("# Core Runtime\n")
3453:         f.write("-r requirements-core.txt\n\n")
3454:
3455:         f.write("# Optional Runtime\n")
3456:         f.write("-r requirements-optional.txt\n\n")
3457:
3458:         f.write("# Development\n")
3459:         for pkg, version in sorted(DEV_DEPENDENCIES.items()):
3460:             f.write(f"{pkg}=={version}\n")
3461:         f.write("\n")
3462:
3463:         f.write("# Documentation\n")
3464:         for pkg, version in sorted(DOCS_DEPENDENCIES.items()):
3465:             f.write(f"{pkg}=={version}\n")
3466:
3467:
3468: def generate_constraints_file(output_file: Path):
3469:     """Generate constraints.txt file."""
3470:     print(f"Generating {output_file}...")
```

```
3471:
3472:     with open(output_file, 'w') as f:
3473:         f.write("# Constraints file - Exact version pins for ALL dependencies\n")
3474:         f.write("# Generated by scripts/generate_dependency_files.py\n")
3475:         f.write("# Use with: pip install -c constraints.txt -r requirements.txt\n\n")
3476:         f.write("# This file prevents dependency conflicts by pinning all versions\n")
3477:         f.write("# including transitive dependencies.\n\n")
3478:         f.write("# To regenerate transitive dependencies:\n")
3479:         f.write("# 1. Install all requirements in a clean venv\n")
3480:         f.write("# 2. Run: pip freeze > constraints-full.txt\n")
3481:         f.write("# 3. Review and merge into this file\n\n")
3482:
3483:         # Combine all dependencies
3484:         all_deps = {}
3485:         all_deps.update(CORE_DEPENDENCIES)
3486:         all_deps.update(OPTIONAL_DEPENDENCIES)
3487:         all_deps.update(DEV_DEPENDENCIES)
3488:         all_deps.update(DOCS_DEPENDENCIES)
3489:
3490:         for pkg, version in sorted(all_deps.items()):
3491:             f.write(f"{pkg}=={version}\n")
3492:
3493:
3494: def main():
3495:     """Main entry point."""
3496:     project_root = Path(__file__).parent.parent
3497:
3498:     # Load existing versions from requirements.txt
3499:     existing_versions = get_current_versions(project_root / "requirements.txt")
3500:
3501:     # Generate all requirement files
3502:     generate_core_requirements(project_root / "requirements-core.txt", existing_versions)
3503:     generate_optional_requirements(project_root / "requirements-optional.txt", existing_versions)
3504:     generate_dev_requirements(project_root / "requirements-dev.txt", existing_versions)
3505:     generate_docs_requirements(project_root / "requirements-docs.txt", existing_versions)
3506:     generate_all_requirements(project_root / "requirements-all.txt", existing_versions)
3507:     generate_constraints_file(project_root / "constraints-new.txt")
3508:
3509:     print("\n" + "="*80)
3510:     print("Dependency files generated successfully!")
3511:     print("="*80)
3512:     print("\nGenerated files:")
3513:     print("  - requirements-core.txt: Core runtime dependencies")
3514:     print("  - requirements-optional.txt: Optional runtime dependencies")
3515:     print("  - requirements-dev.txt: Development dependencies (includes core)")
3516:     print("  - requirements-docs.txt: Documentation dependencies")
3517:     print("  - requirements-all.txt: All dependencies combined")
3518:     print("  - constraints-new.txt: Version constraints for all packages")
3519:     print("\nNext steps:")
3520:     print("  1. Review the generated files")
3521:     print("  2. Test installation: pip install -r requirements-core.txt")
3522:     print("  3. Run verification: make deps:verify")
3523:     print("  4. Replace old files if everything works")
3524:
3525:
3526: if __name__ == "__main__":
```

```
3527:     main()
3528:
3529:
3530:
3531: ================================================================================
3532: FILE: scripts/operations/generate_graphs.py
3533: ================================================================================
3534:
3535: import matplotlib.pyplot as plt
3536: import numpy as np
3537: import argparse
3538: import os
3539:
3540: # AtroZ Dashboard Color Palette
3541: colors = {
3542:     'bg': '#0A0A0A',
3543:     'ink': '#E5E7EB',
3544:     'red': '#C41E3A',
3545:     'blue': '#00D4FF',
3546:     'green': '#39FF14',
3547:     'copper': '#B2642E',
3548:     'copper_oxide': '#17A589'
3549: }
3550:
3551: plt.rcParams['figure.facecolor'] = colors['bg']
3552: plt.rcParams['axes.facecolor'] = colors['bg']
3553: plt.rcParams['text.color'] = colors['ink']
3554: plt.rcParams['axes.labelcolor'] = colors['ink']
3555: plt.rcParams['xtick.color'] = colors['ink']
3556: plt.rcParams['ytick.color'] = colors['ink']
3557: plt.rcParams['axes.edgecolor'] = colors['copper']
3558: plt.rcParams['font.family'] = 'JetBrains Mono'
3559:
3560: def get_text(lang, key):
3561:     """Gets the text for the given language and key."""
3562:     text = {
3563:         'en': {
3564:             'control_flow_title': 'Control-Flow Graph',
3565:             'input_config': 'Input config',
3566:             'schema_validation': 'Schema & path validation',
3567:             'load_questionnaire': 'Load canonical questionnaire',
3568:             'derive_settings': 'Derive AggregationSettings',
3569:             'enforce_graph': 'Enforce phase graph + dependencies',
3570:             'emit_config': 'Emit validated config',
3571:             'reject_run': 'Reject run',
3572:             'fail': 'fail',
3573:             'hash_mismatch': 'hash mismatch',
3574:             'data_flow_title': 'Data-Flow Graph',
3575:             'config_raw': 'ConfigRaw',
3576:             'schema_validator': 'SchemaValidator',
3577:             'loader': 'Loader',
3578:             'hash_verifier': 'HashVerifier',
3579:             'settings_builder': 'SettingsBuilder',
3580:             'config_validated': 'ConfigValidated',
3581:             'questionnaire_file': 'QuestionnaireFile',
3582:             'executor_config': 'ExecutorConfig',
```

```
            'calibration_profiles': 'CalibrationProfiles',
            'dependency_validator': 'DependencyValidator',
            'state_transition_title': 'State-Transition Graph',
            'idle': 'Idle',
            'validating': 'Validating',
            'loading': 'Loading',
            'enforcing_graph': 'EnforcingGraph',
            'dependency_check': 'DependencyCheck',
            'emitting': 'Emitting',
            'faulted': 'Faulted',
            'contract_linkage_title': 'Contract-Linkage Graph'
        },
        'es': {
            'control_flow_title': 'Grafo de Flujo de Control',
            'input_config': 'Config de entrada',
            'schema_validation': 'ValidaciÃ³n de esquema y ruta',
            'load_questionnaire': 'Cargar cuestionario canÃ³nico',
            'derive_settings': 'Derivar AggregationSettings',
            'enforce_graph': 'Hacer cumplir grafo de fases + dependencias',
            'emit_config': 'Emitir config validado',
            'reject_run': 'Rechazar ejecuciÃ³n',
            'fail': 'falla',
            'hash_mismatch': 'hash incorrecto',
            'data_flow_title': 'Grafo de Flujo de Datos',
            'config_raw': 'ConfigRaw',
            'schema_validator': 'ValidadorDeEsquema',
            'loader': 'Cargador',
            'hash_verifier': 'VerificadorDeHash',
            'settings_builder': 'ConstructorDeConfiguracion',
            'config_validated': 'ConfigValidada',
            'questionnaire_file': 'ArchivoDeCuestionario',
            'executor_config': 'ConfigDeEjecutor',
            'calibration_profiles': 'PerfilesDeCalibracion',
            'dependency_validator': 'ValidadorDeDependencias',
            'state_transition_title': 'Grafo de TransiciÃ³n de Estado',
            'idle': 'Inactivo',
            'validating': 'Validando',
            'loading': 'Cargando',
            'enforcing_graph': 'AplicandoGrafo',
            'dependency_check': 'VerificandoDeps',
            'emitting': 'Emitiendo',
            'faulted': 'Fallido',
            'contract_linkage_title': 'Grafo de VÃ-nculos de Contrato'
        }
    }
    return text[lang][key]

def create_control_flow_graph(lang='en'):
    """Generates the control-flow graph."""
    fig, ax = plt.subplots(figsize=(10, 6))
    ax.set_xlim(0, 10)
    ax.set_ylim(0, 6)
    ax.axis('off')

    nodes = {
        'A': (1, 5, get_text(lang, 'input_config')),
```

```
3639:         'B': (3, 5, get_text(lang, 'schema_validation')),
3640:         'C': (5, 5, get_text(lang, 'load_questionnaire')),
3641:         'D': (7, 5, get_text(lang, 'derive_settings')),
3642:         'E': (9, 5, get_text(lang, 'enforce_graph')),
3643:         'F': (7, 3, get_text(lang, 'emit_config')),
3644:         'Z': (5, 1, get_text(lang, 'reject_run'))
3645:     }
3646:
3647:     for node, (x, y, label) in nodes.items():
3648:         ax.text(x, y, label, ha='center', va='center', bbox=dict(boxstyle='round,pad=0.5', fc=colors['bg'], ec=colors['blue']))
3649:
3650:     arrows = [
3651:         ('A', 'B'), ('B', 'C'), ('C', 'D'), ('D', 'E'), ('E', 'F'),
3652:         ('B', 'Z', get_text(lang, 'fail')), ('C', 'Z', get_text(lang, 'hash_mismatch'))
3653:     ]
3654:
3655:     for start, end, *label in arrows:
3656:         x_start, y_start, _ = nodes[start]
3657:         x_end, y_end, _ = nodes[end]
3658:         ax.annotate('', xy=(x_end, y_end), xytext=(x_start, y_start),
3659:                     arrowprops=dict(arrowstyle='->', color=colors['copper'], lw=2))
3660:         if label:
3661:             ax.text((x_start + x_end) / 2, (y_start + y_end) / 2, label[0],
3662:                     ha='center', va='center', color=colors['red'])
3663:
3664:     filename = f'docs/phases/phase_0/images/control_flow_{lang}.png'
3665:     plt.savefig(filename, bbox_inches='tight')
3666:     plt.close()
3667:     print(f"Generated {filename}")
3668:
3669: def create_data_flow_graph(lang='en'):
3670:     """Generates the data-flow graph."""
3671:     fig, ax = plt.subplots(figsize=(12, 4))
3672:     ax.set_xlim(0, 12)
3673:     ax.set_ylim(0, 4)
3674:     ax.axis('off')
3675:
3676:     nodes = {
3677:         'ConfigRaw': (1, 3, get_text(lang, 'config_raw')),
3678:         'SchemaValidator': (3, 3, get_text(lang, 'schema_validator')),
3679:         'Loader': (5, 3, get_text(lang, 'loader')),
3680:         'HashVerifier': (7, 3, get_text(lang, 'hash_verifier')),
3681:         'SettingsBuilder': (9, 3, get_text(lang, 'settings_builder')),
3682:         'ConfigValidated': (11, 3, get_text(lang, 'config_validated')),
3683:         'QuestionnaireFile': (5, 1, get_text(lang, 'questionnaire_file')),
3684:         'ExecutorConfig': (7, 1, get_text(lang, 'executor_config')),
3685:         'CalibrationProfiles': (9, 1, get_text(lang, 'calibration_profiles')),
3686:         'DependencyValidator': (8, 2, get_text(lang, 'dependency_validator'))
3687:     }
3688:
3689:     for node, (x, y, label) in nodes.items():
3690:         ax.text(x, y, label, ha='center', va='center', bbox=dict(boxstyle='round,pad=0.5', fc=colors['bg'], ec=colors['green']))
3691:
3692:     arrows = [
3693:         ('ConfigRaw', 'SchemaValidator'), ('SchemaValidator', 'Loader'),
3694:         ('QuestionnaireFile', 'Loader'), ('Loader', 'HashVerifier'),
```

```
        ('HashVerifier', 'SettingsBuilder'), ('SettingsBuilder', 'ConfigValidated'),
        ('ExecutorConfig', 'DependencyValidator'), ('CalibrationProfiles', 'DependencyValidator'),
        ('DependencyValidator', 'ConfigValidated')
    ]

    for start, end in arrows:
        x_start, y_start, _ = nodes[start]
        x_end, y_end, _ = nodes[end]
        ax.annotate('', xy=(x_end, y_end), xytext=(x_start, y_start),
                    arrowprops=dict(arrowstyle='->', color=colors['copper'], lw=2))

    filename = f'docs/phases/phase_0/images/data_flow_{lang}.png'
    plt.savefig(filename, bbox_inches='tight')
    plt.close()
    print(f"Generated {filename}")


def create_state_transition_graph(lang='en'):
    """Generates the state-transition graph."""
    fig, ax = plt.subplots(figsize=(8, 8), subplot_kw=dict(projection='polar'))
    ax.set_facecolor(colors['bg'])
    ax.grid(color=colors['copper'], linestyle='--', linewidth=0.5)
    ax.spines['polar'].set_edgecolor(colors['copper'])

    states = [
        get_text(lang, 'idle'), get_text(lang, 'validating'), get_text(lang, 'loading'),
        get_text(lang, 'enforcing_graph'), get_text(lang, 'dependency_check'),
        get_text(lang, 'emitting'), get_text(lang, 'faulted')
    ]
    theta = np.linspace(0, 2 * np.pi, len(states), endpoint=False)

    ax.set_xticks(theta)
    ax.set_xticklabels(states)
    ax.set_yticklabels([])

    transitions = [
        (get_text(lang, 'idle'), get_text(lang, 'validating')),
        (get_text(lang, 'validating'), get_text(lang, 'loading')),
        (get_text(lang, 'loading'), get_text(lang, 'enforcing_graph')),
        (get_text(lang, 'enforcing_graph'), get_text(lang, 'dependency_check')),
        (get_text(lang, 'dependency_check'), get_text(lang, 'emitting')),
        (get_text(lang, 'emitting'), get_text(lang, 'idle')),
        (get_text(lang, 'validating'), get_text(lang, 'faulted')),
        (get_text(lang, 'enforcing_graph'), get_text(lang, 'faulted')),
        (get_text(lang, 'dependency_check'), get_text(lang, 'faulted'))
    ]

    for start, end in transitions:
        start_idx = states.index(start)
        end_idx = states.index(end)
        ax.annotate('', xy=(theta[end_idx], 1), xytext=(theta[start_idx], 1),
                    arrowprops=dict(arrowstyle='->', color=colors['red'],
                                    connectionstyle='arc3,rad=0.2'))

    filename = f'docs/phases/phase_0/images/state_transition_{lang}.png'
    plt.savefig(filename, bbox_inches='tight')
```

```
    plt.close()
    print(f"Generated {filename}")

def create_contract_linkage_graph(lang='en'):
    """Generates the contract-linkage graph."""
    fig, ax = plt.subplots(figsize=(10, 8))
    ax.set_xlim(0, 10)
    ax.set_ylim(0, 8)
    ax.axis('off')

    contracts = {
        'C0-CONFIG-V1': (2, 7), 'QMONO-V1': (2, 5), 'HASH-V1': (2, 3),
        'AGG-SET-V1': (2, 1), 'GRAPH-V1': (8, 7), 'EXEC-CONF / CAL-V1': (8, 5)
    }

    validators = {
        get_text(lang, 'schema_validator'): (4, 7),
        get_text(lang, 'loader'): (4, 5),
        get_text(lang, 'hash_verifier'): (4, 3),
        get_text(lang, 'settings_builder'): (4, 1),
        get_text(lang, 'enforcing_graph'): (6, 7),
        get_text(lang, 'dependency_validator'): (6, 5)
    }

    for contract, (x, y) in contracts.items():
        ax.text(x, y, contract, ha='center', va='center', bbox=dict(boxstyle='sawtooth,pad=0.5', fc=colors['bg'], ec=colors['copper_oxide']))

    for validator, (x, y) in validators.items():
        ax.text(x, y, validator, ha='center', va='center', bbox=dict(boxstyle='round,pad=0.5', fc=colors['bg'], ec=colors['blue']))

    arrows = [
        ('C0-CONFIG-V1', get_text(lang, 'schema_validator')),
        ('QMONO-V1', get_text(lang, 'loader')),
        ('HASH-V1', get_text(lang, 'hash_verifier')),
        ('AGG-SET-V1', get_text(lang, 'settings_builder')),
        ('GRAPH-V1', get_text(lang, 'enforcing_graph')),
        ('EXEC-CONF / CAL-V1', get_text(lang, 'dependency_validator'))
    ]

    for start, end in arrows:
        x_start, y_start = contracts.get(start, (0,0))
        x_end, y_end = validators.get(end, (0,0))

        if (x_start,y_start) == (0,0):
             x_start, y_start = validators.get(start, (0,0))
             x_end, y_end = contracts.get(end, (0,0))


        ax.annotate('', xy=(x_end, y_end), xytext=(x_start, y_start),
                    arrowprops=dict(arrowstyle='->', color=colors['copper'], lw=2))

    filename = f'docs/phases/phase_0/images/contract_linkage_{lang}.png'
    plt.savefig(filename, bbox_inches='tight')
    plt.close()
    print(f"Generated {filename}")

```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Generate graphs for the documentation.')
    parser.add_argument('--lang', type=str, default='en', help='The language to generate the graphs in (en or es).')
    args = parser.parse_args()

    output_dir = 'docs/phases/phase_0/images'
    os.makedirs(output_dir, exist_ok=True)

    create_control_flow_graph(args.lang)
    create_data_flow_graph(args.lang)
    create_state_transition_graph(args.lang)
    create_contract_linkage_graph(args.lang)
    print("Graphs generated successfully.")


================================================================================
FILE: scripts/operations/generate_method_classification.py
================================================================================

#!/usr/bin/env python3
"""
Generate Method Classification Artifact for FAKE â\206\222 REAL Executor Migration

This script performs code inspection to classify all methods into three categories:
- REAL_NON_EXEC: Real methods that are not executors (already calibrated, protected)
- FAKE_EXEC: Fake executor methods from old executors.py (invalid, must discard)
- REAL_EXEC: Real executor methods from executors_contract.py (need calibration)

NO placeholders. NO guesswork. Evidence-based classification only.

Output: method_classification.json
"""

from __future__ import annotations

import ast
import json
from pathlib import Path
from typing import Any

PROJECT_ROOT = Path(__file__).resolve().parent.parent
SRC_ROOT = PROJECT_ROOT / "src" / "farfan_pipeline"
ORCHESTRATOR_ROOT = SRC_ROOT / "core" / "orchestrator"


def extract_classes_from_file(file_path: Path) -> list[str]:
    """Extract all class names from a Python file via AST parsing."""
    if not file_path.exists():
        return []

    try:
        source = file_path.read_text(encoding="utf-8")
        tree = ast.parse(source, filename=str(file_path))
    except SyntaxError as exc:
        print(f"WARNING: Could not parse {file_path}: {exc}")
```

```
3863:         return []
3864:
3865:     classes = []
3866:     for node in ast.walk(tree):
3867:         if isinstance(node, ast.ClassDef):
3868:             classes.append(node.name)
3869:
3870:     return classes
3871:
3872:
3873: def get_fake_executors() -> list[str]:
3874:     """Extract FAKE executor class names from old executors.py."""
3875:     fake_file = ORCHESTRATOR_ROOT / "executors.py"
3876:     classes = extract_classes_from_file(fake_file)
3877:
3878:     # Filter to only executor classes (D{n}_Q{m}_* pattern)
3879:     # Exclude BaseExecutor, ExecutorFailure, etc.
3880:     fake_executors = []
3881:     for class_name in classes:
3882:         # Match pattern: D{digit}(_| )Q{digit}_*
3883:         # Examples: D1_Q1_QuantitativeBaselineExtractor, D3_Q2_TargetProportionalityAnalyzer
3884:         if (
3885:             class_name.startswith("D")
3886:             and ("_Q" in class_name or " Q" in class_name)
3887:             and not class_name.startswith("Base")
3888:             and class_name not in ["ExecutorFailure"]
3889:         ):
3890:             # Construct the fully qualified name
3891:             fake_executors.append(f"orchestrator.executors.{class_name}")
3892:
3893:     return sorted(fake_executors)
3894:
3895:
3896: def get_real_executors() -> list[str]:
3897:     """Extract REAL executor class names from executors_contract.py."""
3898:     real_file = ORCHESTRATOR_ROOT / "executors_contract.py"
3899:     classes = extract_classes_from_file(real_file)
3900:
3901:     # All D{n}Q{m}_Executor_Contract classes
3902:     real_executors = []
3903:     for class_name in classes:
3904:         if class_name.endswith("_Executor_Contract") or class_name.endswith("_Executor"):
3905:             real_executors.append(f"orchestrator.executors_contract.{class_name}")
3906:
3907:     # Also add the aliases (D{n}Q{m}_Executor = D{n}Q{m}_Executor_Contract)
3908:     # These are defined in executors_contract.py lines 186-216
3909:     dimension_question_pairs = [
3910:         (d, q) for d in range(1, 7) for q in range(1, 6)
3911:     ]
3912:
3913:     for dim, quest in dimension_question_pairs:
3914:         # Add both the _Contract class and its alias
3915:         contract_class = f"orchestrator.executors_contract.D{dim}Q{quest}_Executor_Contract"
3916:         alias_class = f"orchestrator.executors_contract.D{dim}Q{quest}_Executor"
3917:
3918:         # Only add if not already in the list
```

```
3919:         if contract_class not in real_executors:
3920:             real_executors.append(contract_class)
3921:         if alias_class not in real_executors:
3922:             real_executors.append(alias_class)
3923:
3924:     return sorted(set(real_executors))
3925:
3926:
3927: def load_calibrated_methods() -> set[str]:
3928:     """Load all methods from intrinsic_calibration.json."""
3929:     calibration_file = PROJECT_ROOT / "config" / "intrinsic_calibration.json"
3930:
3931:     if not calibration_file.exists():
3932:         print(f"WARNING: {calibration_file} not found")
3933:         return set()
3934:
3935:     try:
3936:         data = json.loads(calibration_file.read_text(encoding="utf-8"))
3937:     except json.JSONDecodeError as exc:
3938:         print(f"WARNING: Could not parse {calibration_file}: {exc}")
3939:         return set()
3940:
3941:     # Extract method identifiers from the "methods" key
3942:     # Intrinsic calibration uses format: "module.ClassName.method_name"
3943:     methods = set()
3944:     if "methods" in data:
3945:         methods = set(data["methods"].keys())
3946:
3947:     return methods
3948:
3949:
3950: def scan_all_methods() -> set[str]:
3951:     """Scan all Python files in src/farfan_pipeline to find all class.method combinations."""
3952:     all_methods = set()
3953:
3954:     # Walk through all Python files
3955:     for py_file in SRC_ROOT.rglob("*.py"):
3956:         # Skip test files, __pycache__, etc.
3957:         if "__pycache__" in str(py_file) or "test_" in py_file.name:
3958:             continue
3959:
3960:         try:
3961:             source = py_file.read_text(encoding="utf-8")
3962:             tree = ast.parse(source, filename=str(py_file))
3963:         except (SyntaxError, UnicodeDecodeError):
3964:             continue
3965:
3966:         # Extract class names and their methods
3967:         for node in ast.walk(tree):
3968:             if isinstance(node, ast.ClassDef):
3969:                 class_name = node.name
3970:
3971:                 # Get all method names (functions defined in the class)
3972:                 for item in node.body:
3973:                     if isinstance(item, ast.FunctionDef):
3974:                         method_name = item.name
```

```
3975:                         # Skip private methods (but include __init__)
3976:                         if not method_name.startswith("_") or method_name == "__init__":
3977:                             # Use simplified format: ClassName.method_name
3978:                             all_methods.add(f"{class_name}.{method_name}")
3979:
3980:     return all_methods
3981:
3982:
3983: def get_real_non_exec_methods(
3984:     calibrated_methods: set[str],
3985:     fake_executors: list[str],
3986:     real_executors: list[str],
3987: ) -> list[str]:
3988:     """
3989:     Identify REAL_NON_EXEC methods: calibrated methods that are not executors.
3990:
3991:     These are methods that:
3992:     1. Appear in intrinsic_calibration.json
3993:     2. Are NOT fake executors
3994:     3. Are NOT real executors (executors need recalibration)
3995:     """
3996:     # Extract simplified executor names for comparison
3997:     fake_exec_names = set()
3998:     for fake_exec in fake_executors:
3999:         # Extract class name: "orchestrator.executors.D1_Q1_QuantitativeBaselineExtractor"
4000:         # â\206\222 "D1_Q1_QuantitativeBaselineExtractor"
4001:         parts = fake_exec.split(".")
4002:         if len(parts) >= 3:
4003:             fake_exec_names.add(parts[-1])
4004:
4005:         # Also add the alias form used in calibration (D{n}Q{m}_Executor)
4006:         # D1_Q1_QuantitativeBaselineExtractor â\206\222 D1Q1_Executor
4007:         if "_Q" in parts[-1]:
4008:             # Extract D{n}_Q{m} and convert to D{n}Q{m}_Executor
4009:             import re
4010:             match = re.match(r'D(\d+)_Q(\d+)_', parts[-1])
4011:             if match:
4012:                 alias = f"D{match.group(1)}Q{match.group(2)}_Executor"
4013:                 fake_exec_names.add(alias)
4014:
4015:     real_exec_names = set()
4016:     for real_exec in real_executors:
4017:         parts = real_exec.split(".")
4018:         if len(parts) >= 3:
4019:             real_exec_names.add(parts[-1])
4020:
4021:     # Filter calibrated methods
4022:     real_non_exec = []
4023:     fake_exec_methods = []
4024:     real_exec_methods = []
4025:
4026:     for method in calibrated_methods:
4027:         # Method format: "module.ClassName.method_name" or "ClassName.method_name"
4028:         # or "src.module.ClassName.method_name" (full path format)
4029:         parts = method.split(".")
4030:
```

```
4031:         # Extract class name (second-to-last part)
4032:         if len(parts) >= 3:
4033:             class_name = parts[-2]  # ...ClassName.method_name â\206\222 ClassName
4034:         elif len(parts) == 2:
4035:             class_name = parts[0]  # ClassName.method_name â\206\222 ClassName
4036:         else:
4037:             class_name = method  # Just the name
4038:
4039:         # Check if it's from the old executors.py (FAKE)
4040:         is_fake = False
4041:         if "executors." in method and "executors_contract" not in method:
4042:             # Check if it matches executor pattern
4043:             if (
4044:                 class_name.startswith("D")
4045:                 and ("_Q" in class_name or "Q" in class_name)
4046:                 and any(char.isdigit() for char in class_name[:5])  # D{n}_Q{m} or D{n}Q{m}
4047:             ):
4048:                 is_fake = True
4049:                 fake_exec_methods.append(method)
4050:                 continue
4051:
4052:         # Check if it's an executor by name
4053:         if class_name in fake_exec_names or class_name in real_exec_names:
4054:             if class_name in fake_exec_names:
4055:                 fake_exec_methods.append(method)
4056:             else:
4057:                 real_exec_methods.append(method)
4058:             continue
4059:
4060:         # Exclude if it matches executor patterns (D{n}_Q{m} or D{n}Q{m})
4061:         if (
4062:             class_name.startswith("D")
4063:             and ("_Q" in class_name or "Q" in class_name)
4064:             and any(char.isdigit() for char in class_name[:5])  # D{n}_Q{m} or D{n}Q{m}
4065:         ):
4066:             # Likely an executor variant
4067:             fake_exec_methods.append(method)
4068:             continue
4069:
4070:         real_non_exec.append(method)
4071:
4072:     return sorted(real_non_exec), sorted(fake_exec_methods), sorted(real_exec_methods)
4073:
4074:
4075: def generate_classification() -> dict[str, Any]:
4076:     """Generate the complete method classification artifact."""
4077:     print("=" * 80)
4078:     print("GENERATING METHOD CLASSIFICATION ARTIFACT")
4079:     print("=" * 80)
4080:
4081:     print("\n1. Extracting FAKE executors from executors.py...")
4082:     fake_executors = get_fake_executors()
4083:     print(f"   Found {len(fake_executors)} FAKE executor classes")
4084:
4085:     print("\n2. Extracting REAL executors from executors_contract.py...")
4086:     real_executors = get_real_executors()
```

```
    print(f"   Found {len(real_executors)} REAL executor classes")

    print("\n3. Loading calibrated methods from intrinsic_calibration.json...")
    calibrated_methods = load_calibrated_methods()
    print(f"   Found {len(calibrated_methods)} calibrated methods")

    print("\n4. Classifying methods from calibration file...")
    real_non_exec, fake_exec_calibrated, real_exec_calibrated = get_real_non_exec_methods(
        calibrated_methods, fake_executors, real_executors
    )
    print(f"   REAL_NON_EXEC:  {len(real_non_exec):>5} methods (protected)")
    print(f"   FAKE_EXEC:      {len(fake_exec_calibrated):>5} methods in calibration (discard)")
    print(f"   REAL_EXEC:      {len(real_exec_calibrated):>5} methods in calibration (recalibrate)")

    # Construct the artifact
    classification = {
        "_metadata": {
            "version": "1.0",
            "date": "2025-11-24",
            "migration": "FAKE â\206\222 REAL Executor Migration",
            "branch": "claude/fake-real-executor-migration-01DkQrq2dtSN3scUvzNVKqGy",
            "description": (
                "Machine-readable classification of all methods for executor migration. "
                "REAL_NON_EXEC methods have protected calibrations. "
                "FAKE_EXEC methods have invalid calibrations (must discard). "
                "REAL_EXEC methods need new calibrations (all 8 layers)."
            ),
        },
        "REAL_NON_EXEC": {
            "description": "Real methods that are not executors. Already calibrated via rubric. PROTECTED - DO NOT MODIFY.",
            "count": len(real_non_exec),
            "methods": real_non_exec,
        },
        "FAKE_EXEC": {
            "description": "Fake executor methods from old executors.py. Hardcoded execute() methods. INVALID - DISCARD CALIBRATIONS.",
            "count": len(fake_executors),
            "file": "src/farfan_pipeline/core/orchestrator/executors.py",
            "snapshot": "src/farfan_pipeline/core/orchestrator/executors_snapshot/executors.py",
            "classes": fake_executors,
            "calibrated_methods": {
                "count": len(fake_exec_calibrated),
                "status": "INVALID - placeholder_computed",
                "action": "DISCARD",
                "methods": fake_exec_calibrated,
            },
        },
        "REAL_EXEC": {
            "description": "Real executor methods from executors_contract.py. Contract-driven routing. NEED CALIBRATION - ALL 8 LAYERS.",
            "count": len(real_executors),
            "file": "src/farfan_pipeline/core/orchestrator/executors_contract.py",
            "classes": real_executors,
            "calibrated_methods": {
                "count": len(real_exec_calibrated),
                "status": "partial or none",
                "action": "RECALIBRATE",
                "methods": real_exec_calibrated,
```

```
            },
        },
        "summary": {
            "total_classes": len(real_non_exec) + len(fake_executors) + len(real_executors),
            "total_calibrated_methods": len(calibrated_methods),
            "real_non_exec_methods": len(real_non_exec),
            "fake_exec_classes": len(fake_executors),
            "fake_exec_calibrated_methods": len(fake_exec_calibrated),
            "real_exec_classes": len(real_executors),
            "real_exec_calibrated_methods": len(real_exec_calibrated),
        },
    }

    return classification


def main() -> None:
    """Main entry point."""
    classification = generate_classification()

    # Write to file
    output_file = PROJECT_ROOT / "method_classification.json"
    with output_file.open("w", encoding="utf-8") as f:
        json.dump(classification, f, indent=2, ensure_ascii=False)

    print(f"\n{'=' * 80}")
    print(f"â\234\205 Classification artifact written to: {output_file}")
    print(f"{'=' * 80}")
    print("\nSUMMARY:")
    print(f"  REAL_NON_EXEC:             {classification['summary']['real_non_exec_methods']:>5} methods (protected)")
    print(f"  FAKE_EXEC classes:         {classification['summary']['fake_exec_classes']:>5} classes")
    print(f"    - in calibration file:   {classification['summary']['fake_exec_calibrated_methods']:>5} methods (DISCARD)")
    print(f"  REAL_EXEC classes:         {classification['summary']['real_exec_classes']:>5} classes")
    print(f"    - in calibration file:   {classification['summary']['real_exec_calibrated_methods']:>5} methods (recalibrate)")
    print(f"  {'â\224\200' * 50}")
    print(f"  Total calibrated methods:  {classification['summary']['total_calibrated_methods']:>5}")
    print()


if __name__ == "__main__":
    main()



================================================================================
FILE: scripts/operations/update_questionnaire_metadata.py
================================================================================

#!/usr/bin/env python3
"""Utility to update questionnaire metadata with specificity and dependencies."""
from __future__ import annotations

import json
from pathlib import Path
from typing import Any

```

```
4199: ROOT = Path(__file__).resolve().parents[1]
4200: # UPDATED: cuestionario_FIXED.json migrated to questionnaire_monolith.json
4201: QUESTIONNAIRE_FILES = [
4202:     ROOT / "data" / "questionnaire_monolith.json",
4203: ]
4204:
4205: SPECIFICITY_HIGH_KEYWORDS = {
4206:     "cuant",  # cuantificacion, cuantitativo
4207:     "magnitud",
4208:     "brecha",
4209:     "trazabilidad",
4210:     "asignacion",
4211:     "coherencia",
4212:     "proporcional",
4213:     "meta",
4214:     "impacto",
4215:     "resultado",
4216:     "suficiencia",
4217:     "evidencia",
4218:     "ambicion",
4219:     "contradiccion",
4220:     "cobertura",
4221:     "linea_base",
4222: }
4223:
4224: SPECIFICITY_MEDIUM_KEYWORDS = {
4225:     "vacio",
4226:     "vacÃ-o",
4227:     "limit",
4228:     "sesgo",
4229:     "riesgo",
4230:     "particip",
4231:     "proceso",
4232:     "gobernanza",
4233:     "articulacion",
4234:     "coordinacion",
4235:     "capacidad",
4236:     "enfoque",
4237:     "seguimiento",
4238:     "soporte",
4239: }
4240:
4241: SPECIFICITY_LEVELS = ("HIGH", "MEDIUM", "LOW")
4242:
4243: SCORING_LEVELS = ["excelente", "bueno", "aceptable", "insuficiente"]
4244:
4245: DEFAULT_SCORING = {
4246:     "excelente": {"min_score": 0.85, "criteria": ""},
4247:     "bueno": {"min_score": 0.7, "criteria": ""},
4248:     "aceptable": {"min_score": 0.55, "criteria": ""},
4249:     "insuficiente": {"min_score": 0.0, "criteria": ""},
4250: }
4251:
4252: DEPENDENCIAS_MAP = {
4253:     "D3-Q2": {
4254:         "brecha_diagnosticada": "D1-Q2",
```

```
        "recursos_asignados": "D1-Q3",
    },
    "D4-Q3": {
        "inversion_total": "D1-Q3",
        "capacidad_mencionada": "D1-Q4",
    },
}

def assign_specificity(group_name: str) -> str:
    name = group_name.lower()
    if any(keyword in name for keyword in SPECIFICITY_HIGH_KEYWORDS):
        return "HIGH"
    if any(keyword in name for keyword in SPECIFICITY_MEDIUM_KEYWORDS):
        return "MEDIUM"
    return "MEDIUM"

def normalize_scoring(scoring: dict[str, Any]) -> dict[str, Any]:
    normalized: dict[str, Any] = {}
    for level in SCORING_LEVELS:
        entry = scoring.get(level, {})
        entry_dict = dict(entry) if isinstance(entry, dict) else {}
        # Preserve existing values but guarantee required keys
        if "min_score" not in entry_dict:
            entry_dict["min_score"] = DEFAULT_SCORING[level]["min_score"]
        if "criteria" not in entry_dict:
            entry_dict["criteria"] = DEFAULT_SCORING[level]["criteria"]
        normalized[level] = entry_dict
    # Append any additional scoring categories after the normalized block
    for level, entry in scoring.items():
        if level not in normalized:
            normalized[level] = entry
    return normalized

def update_verification_blocks(question: dict[str, Any]) -> bool:
    updated = False
    for key, value in list(question.items()):
        if not key.startswith("verificacion"):
            continue
        if isinstance(value, dict):
            for group_name, group_data in value.items():
                if isinstance(group_data, dict) and "patterns" in group_data:
                    specificity = assign_specificity(group_name)
                    if group_data.get("specificity") != specificity:
                        group_data["specificity"] = specificity
                        updated = True
    return updated

def apply_updates(path: Path) -> bool:
    if not path.exists():
        return False
    changed = False
    data = json.loads(path.read_text(encoding="utf-8"))

    preguntas = data.get("preguntas_base", [])
    if isinstance(preguntas, list):
        for question in preguntas:
```

```
            if not isinstance(question, dict):
                continue
            # Update verification specificity
            if update_verification_blocks(question):
                changed = True
            # Normalize scoring structure
            scoring = question.get("scoring")
            if isinstance(scoring, dict):
                normalized = normalize_scoring(scoring)
                if normalized != scoring:
                    question["scoring"] = normalized
                    changed = True
            # Apply dependencies if applicable
            metadata = question.get("metadata", {})
            original_id = metadata.get("original_id") if isinstance(metadata, dict) else None
            if original_id in DEPENDENCIAS_MAP:
                deps = DEPENDENCIAS_MAP[original_id]
                if question.get("dependencias_data") != deps:
                    question["dependencias_data"] = deps
                    changed = True
    if changed:
        path.write_text(json.dumps(data, ensure_ascii=False, indent=2) + "\n", encoding="utf-8")
    return changed

def main() -> None:
    any_changed = False
    for file_path in QUESTIONNAIRE_FILES:
        if apply_updates(file_path):
            print(f"Updated {file_path.relative_to(ROOT)}")
            any_changed = True
        else:
            print(f"No changes required for {file_path.relative_to(ROOT)}")
    if not any_changed:
        print("No updates were necessary")

if __name__ == "__main__":
    main()



================================================================================
FILE: scripts/pipeline/run_complete_analysis_plan1.py
================================================================================

#!/usr/bin/env python3
"""Complete System Execution: SPC + Orchestrator for Plan_1.pdf

This script demonstrates the complete end-to-end processing pipeline:
1. CPP Ingestion: Preprocess Plan_1.pdf using Canon Policy Package pipeline
2. SPC Adaptation: Convert CPP to PreprocessedDocument format using SPCAdapter
3. Orchestrator Execution: Run all 11 phases of the orchestration pipeline
4. Results Display: Show comprehensive results from each phase

Usage:
    python run_complete_analysis_plan1.py

```

```
Requirements:
    - Plan_1.pdf must exist in data/plans/
    - All dependencies installed (pdfplumber, pyarrow, etc.)

Note: Run this script after installing the package with: pip install -e .
"""

import asyncio
import sys
import uuid
from datetime import datetime
from pathlib import Path

from farfan_pipeline.utils.paths import data_dir
from farfan_pipeline.processing.spc_ingestion import CPPIngestionPipeline  # Updated to SPC ingestion
from farfan_pipeline.utils.spc_adapter import SPCAdapter
from farfan_pipeline.core.orchestrator import Orchestrator
from farfan_pipeline.core.orchestrator.factory import build_processor
from farfan_pipeline.processing.cpp_ingestion.models import CanonPolicyPackage
from farfan_pipeline.utils.proof_generator import (
    ProofData,
    compute_code_signatures,
    compute_dict_hash,
    compute_file_hash,
    verify_success_conditions,
    generate_proof,
    collect_artifacts_manifest,
)
from farfan_pipeline.core.runtime_config import RuntimeConfig
from farfan_pipeline.core.boot_checks import run_boot_checks, get_boot_check_summary, BootCheckError
from farfan_pipeline.core.observability.structured_logging import log_runtime_config_loaded


def load_cpp_from_directory(cpp_dir: Path) -> CanonPolicyPackage:
    """
    Load Canon Policy Package from a directory with Arrow files and metadata.

    Args:
        cpp_dir: Directory containing CPP files (content_stream.arrow, etc.)

    Returns:
        Reconstructed CanonPolicyPackage
    """
    import json
    import pyarrow as pa
    import pyarrow.ipc as ipc
    from farfan_pipeline.processing.cpp_ingestion.models import (
        CanonPolicyPackage,
        ChunkGraph,
        IntegrityIndex,
        PolicyManifest,
        ProvenanceMap,
        QualityMetrics,
    )

    # Load metadata
```

```
4423:     metadata_path = cpp_dir / "metadata.json"
4424:     with open(metadata_path, 'r') as f:
4425:         metadata = json.load(f)
4426:
4427:     # Load content stream
4428:     content_stream = None
4429:     content_stream_path = cpp_dir / "content_stream.arrow"
4430:     if content_stream_path.exists():
4431:         with pa.OSFile(str(content_stream_path), "rb") as source:
4432:             with ipc.open_file(source) as reader:
4433:                 content_stream = reader.read_all()
4434:
4435:     # Load provenance map
4436:     provenance_table = None
4437:     provenance_path = cpp_dir / "provenance_map.arrow"
4438:     if provenance_path.exists():
4439:         with pa.OSFile(str(provenance_path), "rb") as source:
4440:             with ipc.open_file(source) as reader:
4441:                 provenance_table = reader.read_all()
4442:
4443:     # Reconstruct objects
4444:     policy_manifest = PolicyManifest(
4445:         axes=metadata["policy_manifest"]["axes"],
4446:         programs=metadata["policy_manifest"]["programs"],
4447:         projects=[],
4448:         years=metadata["policy_manifest"]["years"],
4449:         territories=metadata["policy_manifest"]["territories"],
4450:         indicators=[],
4451:         budget_rows=[],
4452:     )
4453:
4454:     integrity_index = IntegrityIndex(
4455:         blake3_root=metadata["integrity_index"]["blake3_root"],
4456:         chunk_hashes={},
4457:     )
4458:
4459:     quality_metrics = QualityMetrics(
4460:         boundary_f1=metadata["quality_metrics"]["boundary_f1"],
4461:         kpi_linkage_rate=metadata["quality_metrics"]["kpi_linkage_rate"],
4462:         budget_consistency_score=metadata["quality_metrics"]["budget_consistency_score"],
4463:         provenance_completeness=1.0,
4464:         structural_consistency=1.0,
4465:         temporal_robustness=1.0,
4466:         chunk_context_coverage=1.0,
4467:     )
4468:
4469:     provenance_map = ProvenanceMap(table=provenance_table)
4470:
4471:     # Create chunks from content stream
4472:     # Since chunk_graph isn't saved separately, we reconstruct minimal chunks from content_stream
4473:     from farfan_pipeline.processing.cpp_ingestion.models import (
4474:         Chunk, ChunkResolution, TextSpan, Confidence,
4475:         PolicyFacet, TimeFacet, GeoFacet
4476:     )
4477:
4478:     chunks = {}
```

```
4479:     if content_stream is not None:
4480:         for i in range(content_stream.num_rows):
4481:             row = content_stream.slice(i, 1)
4482:             page_id = row.column("page_id")[0].as_py()
4483:             text = row.column("text")[0].as_py()
4484:             byte_start = row.column("byte_start")[0].as_py()
4485:             byte_end = row.column("byte_end")[0].as_py()
4486:
4487:             # Create a minimal chunk with all required facets
4488:             chunk_id = f"chunk_{i}"
4489:             chunks[chunk_id] = Chunk(
4490:                 id=chunk_id,
4491:                 text=text,
4492:                 resolution=ChunkResolution.MESO,  # Default to MESO
4493:                 text_span=TextSpan(start=byte_start, end=byte_end),
4494:                 bytes_hash=f"hash_{i}",  # Placeholder
4495:                 policy_facets=PolicyFacet(),  # Empty policy facets
4496:                 time_facets=TimeFacet(),  # Empty time facets
4497:                 geo_facets=GeoFacet(),  # Empty geo facets
4498:                 provenance=None,
4499:                 kpi=None,
4500:                 budget=None,
4501:                 entities=[],
4502:                 confidence=Confidence(layout=1.0, ocr=1.0, typing=1.0),
4503:             )
4504:
4505:     chunk_graph = ChunkGraph(chunks=chunks)
4506:
4507:     # Create CPP
4508:     cpp = CanonPolicyPackage(
4509:         schema_version=metadata["schema_version"],
4510:         policy_manifest=policy_manifest,
4511:         chunk_graph=chunk_graph,
4512:         content_stream=content_stream,
4513:         provenance_map=provenance_map,
4514:         integrity_index=integrity_index,
4515:         quality_metrics=quality_metrics,
4516:     )
4517:
4518:     return cpp
4519:
4520:
4521: async def main():
4522:     """Main execution function."""
4523:
4524:     # ============================================================================
4525:     # PHASE 0: RUNTIME CONFIGURATION & BOOT CHECKS
4526:     # ============================================================================
4527:     print("=" * 80)
4528:     print("F.A.R.F.A.N COMPLETE ANALYSIS PIPELINE")
4529:     print("=" * 80)
4530:     print()
4531:
4532:     print("â\232\231ï¸\217  PHASE 0: RUNTIME CONFIGURATION")
4533:     print("-" * 80)
4534:
```

```
4535:     # Initialize runtime configuration
4536:     runtime_config = RuntimeConfig.from_env()
4537:     print(f"  â\234\223 Runtime mode: {runtime_config.mode.value}")
4538:     print(f"  â\234\223 Strict mode: {runtime_config.is_strict_mode()}")
4539:     print(f"  â\234\223 Preferred spaCy model: {runtime_config.preferred_spacy_model}")
4540:     print()
4541:
4542:     # Log runtime config
4543:     log_runtime_config_loaded(
4544:         config_repr=repr(runtime_config),
4545:         runtime_mode=runtime_config.mode
4546:     )
4547:
4548:     # Run boot checks
4549:     print("ð\237\224\215 BOOT CHECKS")
4550:     print("-" * 80)
4551:     try:
4552:         boot_results = run_boot_checks(runtime_config)
4553:         boot_summary = get_boot_check_summary(boot_results)
4554:         print(boot_summary)
4555:         print()
4556:     except BootCheckError as e:
4557:         print(f"\nâ\235\214 FATAL: Boot check failed: {e}")
4558:         print(f"   Component: {e.component}")
4559:         print(f"   Code: {e.code}")
4560:         print(f"   Reason: {e.reason}")
4561:         if runtime_config.mode.value == "prod":
4562:             print("\n   Aborting execution in PROD mode.\n")
4563:             return 1
4564:         else:
4565:             print(f"\n   â\232 ï¸\217  Continuing in {runtime_config.mode.value} mode despite failure.\n")
4566:
4567:     print("=" * 80)
4568:     print("CPP + ORCHESTRATOR PIPELINE: Plan_1.pdf")
4569:     print("=" * 80)
4570:     print()
4571:
4572:     # ============================================================================
4573:     # PHASE 1: CPP INGESTION
4574:     # ============================================================================
4575:     print("ð\237\223\204 PHASE 1: CPP INGESTION")
4576:     print("-" * 80)
4577:
4578:     input_path = data_dir() / 'plans' / 'Plan_1.pdf'
4579:     cpp_output = data_dir() / 'output' / 'cpp_plan_1'
4580:     cpp_output.mkdir(parents=True, exist_ok=True)
4581:
4582:     if not input_path.exists():
4583:         print(f"â\235\214 ERROR: Plan_1.pdf not found at {input_path}")
4584:         print("   Please ensure the file exists before running.")
4585:         return 1
4586:
4587:     print(f'  Input: Plan_1.pdf')
4588:     print(f'  Location: {input_path}')
4589:     print(f'  Size: {input_path.stat().st_size / 1024:.1f} KB')
4590:     print()
```

```
4591:
4592:     print('  ð\237\224\204 Initializing SPC ingestion pipeline (canonical phase-one)...')
4593:     # Updated to use SPC API (Smart Policy Chunks)
4594:     cpp_pipeline = CPPIngestionPipeline(questionnaire_path=None)  # Uses canonical path
4595:
4596:     print('  ð\237\224\204 Processing document (this may take 30-60 seconds)...')
4597:     # Note: .process() is async and returns CanonPolicyPackage directly
4598:     cpp = await cpp_pipeline.process(
4599:         document_path=input_path,
4600:         document_id='Plan_1',
4601:         title='Plan_1',
4602:         max_chunks=50
4603:     )
4604:
4605:     if not cpp:
4606:         print(f'  â\235\214 SPC Ingestion FAILED: No package returned')
4607:         return 1
4608:
4609:     print(f'  â\234\205 SPC Ingestion completed successfully')
4610:     print(f'  â\234\205 Chunks generated: {len(cpp.chunk_graph.chunks) if cpp.chunk_graph else 0}')
4611:     print(f'  â\234\205 Schema Version: v3.0 (SPC)')
4612:     print()
4613:
4614:     # ============================================================================
4615:     # PHASE 2: SPC ADAPTATION
4616:     # ============================================================================
4617:     print("ð\237\224\204 PHASE 2: SPC ADAPTATION")
4618:     print("-" * 80)
4619:
4620:     print('  ð\237\224\204 Converting CanonPolicyPackage to PreprocessedDocument...')
4621:     adapter = SPCAdapter()
4622:     preprocessed_doc = adapter.to_preprocessed_document(
4623:         cpp,
4624:         document_id='Plan_1'
4625:     )
4626:
4627:     print(f'  â\234\205 Document ID: {preprocessed_doc.document_id}')
4628:     print(f'  â\234\205 Sentences: {len(preprocessed_doc.sentences)}')
4629:     print(f'  â\234\205 Tables: {len(preprocessed_doc.tables)}')
4630:     print(f'  â\234\205 Raw text length: {len(preprocessed_doc.raw_text)} chars')
4631:
4632:     provenance_completeness = preprocessed_doc.metadata.get('provenance_completeness', 0.0)
4633:     print(f'  â\234\205 Provenance completeness: {provenance_completeness:.2%}')
4634:     print()
4635:
4636:     # ============================================================================
4637:     # PHASE 3: ORCHESTRATOR INITIALIZATION (using official API)
4638:     # ============================================================================
4639:     print("â\232\231ï¸\217  PHASE 3: ORCHESTRATOR INITIALIZATION")
4640:     print("-" * 80)
4641:
4642:     print('  ð\237\224\204 Building processor bundle with build_processor()...')
4643:
4644:     try:
4645:         # Use official API: build_processor() to get processor bundle
4646:         processor_bundle = build_processor()
```

```
        print(f'  â\234\205 Processor bundle created')
        print(f'  â\234\205 Method executor: {type(processor_bundle.method_executor).__name__}')
        print(f'  â\234\205 Questionnaire loaded: {len(processor_bundle.questionnaire)} keys')
        print(f'  â\234\205 Factory catalog loaded: {len(processor_bundle.factory.catalog)} keys')
        print()

        print('  ð\237\224\204 Initializing Orchestrator with official arguments...')
        # Use official API: Orchestrator(monolith=questionnaire, catalog=factory.catalog)
        orchestrator = Orchestrator(
            monolith=processor_bundle.questionnaire,
            catalog=processor_bundle.factory.catalog
        )
        print(f'  â\234\205 Orchestrator initialized')
        print(f'  â\234\205 Phases: {len(orchestrator.FASES)}')
        print(f'  â\234\205 Executors registered: {len(orchestrator.executors)}')
        print()
    except Exception as e:
        print(f'  â\235\214 Failed to initialize orchestrator: {e}')
        print(f'  â\204¹ï¸\217  Error details:')
        import traceback
        traceback.print_exc()
        print()
        return 1

    # ======================================================================
    # PHASE 4: ORCHESTRATOR EXECUTION (11 PHASES)
    # ======================================================================
    print("ð\237\232\200 PHASE 4: ORCHESTRATOR EXECUTION (11 PHASES)")
    print("=" * 80)
    print()

    # Create a temporary PDF path for the orchestrator
    # (it expects a PDF path even though we're providing preprocessed_document)
    temp_pdf_path = str(input_path)

    print('  ð\237\224\204 Starting 11-phase orchestration...')
    print()

    try:
        # Run the complete orchestration pipeline
        phase_results = await orchestrator.process_development_plan_async(
            pdf_path=temp_pdf_path,
            preprocessed_document=preprocessed_doc
        )

        print()
        print("=" * 80)
        print("ð\237\223\212 ORCHESTRATION RESULTS")
        print("=" * 80)
        print()

        # Display results for each phase
        for i, result in enumerate(phase_results):
            phase_label = orchestrator.FASES[i][3] if i < len(orchestrator.FASES) else f"Phase {i}"
            status_icon = "â\234\205" if result.success else "â\235\214"

```

```
            print(f"{status_icon} {phase_label}")
            print(f"   Duration: {result.duration_ms:.0f}ms")
            print(f"   Mode: {result.mode}")

            if result.success and result.data is not None:
                # Show data summary based on phase
                if isinstance(result.data, list):
                    print(f"   Results: {len(result.data)} items")
                elif isinstance(result.data, dict):
                    print(f"   Results: {len(result.data)} keys")
                else:
                    print(f"   Results: {type(result.data).__name__}")

            if result.error:
                print(f"   â\235\214 Error: {result.error}")

            if result.aborted:
                print(f"   â\232 ï¸\217  Aborted")
                break

            print()

        # Summary statistics
        successful = sum(1 for r in phase_results if r.success)
        total = len(phase_results)
        total_time = sum(r.duration_ms for r in phase_results)

        print("=" * 80)
        print("ð\237\223\210 SUMMARY")
        print("=" * 80)
        print(f"  Phases completed: {successful}/{total}")
        print(f"  Total time: {total_time/1000:.1f}s")
        print(f"  Average per phase: {total_time/total:.0f}ms")

        # ============================================================================
        # PHASE 5: CRYPTOGRAPHIC PROOF GENERATION (ONLY ON SUCCESS)
        # ============================================================================
        abort_active = orchestrator.abort_signal.is_aborted()

        # Check if we should generate proof
        success_conditions_met, errors = verify_success_conditions(
            phase_results=phase_results,
            abort_active=abort_active,
            output_dir=cpp_output,
        )

        if success_conditions_met and successful == total:
            print()
            print("=" * 80)
            print("ð\237\224\220 PHASE 5: CRYPTOGRAPHIC PROOF GENERATION")
            print("=" * 80)
            print()

            try:
                # Collect data for proof
                print("  ð\237\224\204 Collecting proof data...")
```

```
4759:
4760:                 # Compute code signatures
4761:                 src_root = Path(__file__).parent / "src" / "farfan_pipeline"
4762:                 code_signatures = compute_code_signatures(src_root)
4763:                 print(f"  â\234\205 Code signatures: {list(code_signatures.keys())}")
4764:
4765:                 # Compute input PDF hash
4766:                 input_pdf_hash = compute_file_hash(input_path)
4767:                 print(f"  â\234\205 Input PDF hash: {input_pdf_hash[:16]}...")
4768:
4769:                 # Compute questionnaire/catalog hashes
4770:                 monolith_hash = compute_dict_hash(processor_bundle.questionnaire)
4771:                 catalog_hash = compute_dict_hash(processor_bundle.factory.catalog)
4772:                 print(f"  â\234\205 Monolith hash: {monolith_hash[:16]}...")
4773:                 print(f"  â\234\205 Catalog hash: {catalog_hash[:16]}...")
4774:
4775:                 # FIXME(PROOF): method_map not directly accessible from processor_bundle
4776:                 # method_map must be derived from real execution data
4777:                 method_map = getattr(processor_bundle, "method_map", None)
4778:                 if method_map is None:
4779:                     # FIXME(PROOF): method_map not exposed by ProcessorBundle; proof must not be generated without it
4780:                     raise RuntimeError("Proof generation aborted: real method_map is unavailable")
4781:                 method_map_hash = compute_dict_hash(method_map)
4782:
4783:                 # Count questions from questionnaire monolith
4784:                 questions_total = 0
4785:                 if 'blocks' in processor_bundle.questionnaire:
4786:                     blocks = processor_bundle.questionnaire['blocks']
4787:                     if 'micro_questions' in blocks and isinstance(blocks['micro_questions'], list):
4788:                         questions_total = len(blocks['micro_questions'])
4789:
4790:                 # Count questions answered (from micro_questions phase - Phase 2)
4791:                 # Find the micro questions phase by name instead of hardcoded index
4792:                 questions_answered = 0
4793:                 micro_phase_result = None
4794:                 for i, result in enumerate(phase_results):
4795:                     if i < len(orchestrator.FASES):
4796:                         phase_name = orchestrator.FASES[i][3]
4797:                         if "Micro Preguntas" in phase_name or "FASE 2" in phase_name:
4798:                             micro_phase_result = result
4799:                             break
4800:
4801:                 if micro_phase_result and micro_phase_result.data:
4802:                     if isinstance(micro_phase_result.data, list):
4803:                         # Count successful executions (no error)
4804:                         questions_answered = sum(
4805:                             1 for item in micro_phase_result.data
4806:                             if hasattr(item, 'error') and item.error is None
4807:                         )
4808:
4809:                 # Count evidence records from all phases
4810:                 evidence_records = 0
4811:                 for result in phase_results:
4812:                     if not result.data:
4813:                         continue
4814:
```

```
4815:                     # Count items with evidence attribute
4816:                     if isinstance(result.data, list):
4817:                         evidence_records += sum(
4818:                             1 for item in result.data
4819:                             if hasattr(item, 'evidence') and item.evidence is not None
4820:                         )
4821:                     # Some phases may have dict results with evidence
4822:                     elif isinstance(result.data, dict):
4823:                         if 'evidence' in result.data and result.data['evidence']:
4824:                             evidence_records += 1
4825:
4826:                 # Collect artifacts manifest
4827:                 print("  ð\237\224\204 Computing artifact hashes...")
4828:                 artifacts_manifest = collect_artifacts_manifest(cpp_output)
4829:                 print(f"  â\234\205 Artifacts found: {len(artifacts_manifest)}")
4830:
4831:                 # Build proof data
4832:                 proof_data = ProofData(
4833:                     run_id=str(uuid.uuid4()),
4834:                     timestamp_utc=datetime.utcnow().isoformat() + 'Z',
4835:                     phases_total=total,
4836:                     phases_success=successful,
4837:                     questions_total=questions_total,
4838:                     questions_answered=questions_answered,
4839:                     evidence_records=evidence_records,
4840:                     monolith_hash=monolith_hash,
4841:                     questionnaire_hash=monolith_hash,  # Same as monolith for now
4842:                     catalog_hash=catalog_hash,
4843:                     method_map_hash=method_map_hash,
4844:                     code_signature=code_signatures,
4845:                     input_pdf_hash=input_pdf_hash,
4846:                     artifacts_manifest=artifacts_manifest,
4847:                     execution_metadata={
4848:                         'total_duration_ms': total_time,
4849:                         'avg_phase_duration_ms': total_time / total if total > 0 else 0,
4850:                         'input_file': str(input_path),
4851:                         'output_dir': str(cpp_output),
4852:                     }
4853:                 )
4854:
4855:                 # Generate proof files
4856:                 print("  ð\237\224\204 Generating proof.json and proof.hash...")
4857:                 proof_json_path, proof_hash_path = generate_proof(
4858:                     proof_data=proof_data,
4859:                     output_dir=cpp_output,
4860:                 )
4861:
4862:                 print()
4863:                 print(f"  â\234\205 Proof generated: {proof_json_path}")
4864:                 print(f"  â\234\205 Hash generated: {proof_hash_path}")
4865:                 print()
4866:                 print("  ð\237\223\213 Verification instructions:")
4867:                 print(f"     1. cat {proof_json_path}")
4868:                 print(f"     2. cat {proof_hash_path}")
4869:                 print(f"     3. Recompute hash and compare")
4870:                 print()
```

```
4871:
4872:             except Exception as proof_error:
4873:                 print()
4874:                 print(f"  â\232 ï¸\217  Proof generation failed: {proof_error}")
4875:                 import traceback
4876:                 traceback.print_exc()
4877:                 print()
4878:                 print("  â\204¹ï¸\217  Pipeline succeeded but proof generation failed")
4879:                 print()
4880:
4881:             print()
4882:             print("â\234\205 ALL PHASES COMPLETED")
4883:             return 0
4884:         else:
4885:             print()
4886:             print("  â\232 ï¸\217  Some phases failed or were aborted")
4887:             if errors:
4888:                 print("  â\235\214 Proof NOT generated due to:")
4889:                 for error in errors:
4890:                     print(f"     - {error}")
4891:             return 1
4892:
4893:     except Exception as e:
4894:         print()
4895:         print(f"â\235\214 ORCHESTRATION FAILED: {e}")
4896:         import traceback
4897:         traceback.print_exc()
4898:         return 1
4899:
4900:
4901: if __name__ == "__main__":
4902:     exit_code = asyncio.run(main())
4903:     sys.exit(exit_code)
4904:
4905:
4906:
4907: ================================================================================
4908: FILE: scripts/pipeline/run_policy_pipeline_verified.py
4909: ================================================================================
4910:
4911: #!/usr/bin/env python3
4912: """
4913: Compatibility wrapper that delegates to ``farfan_pipeline.scripts.run_policy_pipeline_verified``.
4914:
4915: The real implementation now lives inside the package so the entrypoint
4916: can be invoked via ``python -m farfan_pipeline.scripts.run_policy_pipeline_verified``.
4917: """
4918:
4919: from __future__ import annotations
4920:
4921: from farfan_pipeline.scripts.run_policy_pipeline_verified import cli
4922:
4923:
4924: if __name__ == "__main__":
4925:     cli()
4926:
```

4927: