```
   - Action: This is acceptable but consider migrating away from cpp_adapter.py

2. **For Future Changes**:
   - Avoid importing at module-level in adapter files
   - Use delayed imports (inside functions/methods) if needed
   - Consider factory patterns to break circular dependencies

3. **Testing Recommendation**:
   - Run 'import saaaaaa.utils.spc_adapter' in clean Python process
   - Run 'import saaaaaa.utils.cpp_adapter' in clean Python process
   - Verify no ImportError or circular dependency errors occur
""")

    print("=" * 80)
    print("Analysis complete.")


if __name__ == '__main__':
    main()

===== FILE: scripts/architecture_enforcement_audit.py =====
#!/usr/bin/env python3
"""
Questionnaire Architecture Enforcement Audit Tool

This tool performs comprehensive static analysis to enforce the questionnaire access
architecture:
1. QuestionnaireResourceProvider is the ONLY source for pattern/validation logic
2. factory.py is the ONLY module that may read questionnaire_monolith.json
3. core.py receives QRP via dependency injection
4. arg_router_extended.py and evidence_registry.py must NOT import QRP or read
questionnaire files
"""

import ast
import json
import sys
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

# Repository root
REPO_ROOT = Path(__file__).parent

# Allowed modules for questionnaire access
ALLOWED_QRP_IMPORTERS = {
    "questionnaire_resource_provider.py",  # Self
    "factory.py",  # I/O boundary
    "core.py",  # Via DI
    "__init__.py",  # Package initialization
    "core_module_factory.py",  # Factory for module DI
    "bootstrap.py",  # Wiring initialization
}

# Test files are allowed to import anything they test
def is_test_file(filepath: str) -> bool:
    """Check if file is a test file"""
    return filepath.startswith('tests/') or '/tests/' in filepath or '/test/' in filepath
or Path(filepath).name.startswith('test_')

ALLOWED_MONOLITH_READERS = {
    "factory.py",  # Only factory may read
}

# Violation types
@dataclass
class Violation:
    """A detected architectural violation"""
```

```
    file_path: str
    line_number: int
    violation_type: str
    code_snippet: str
    explanation: str

@dataclass
class AnalysisReport:
    """Complete analysis report"""
    violations: list[Violation] = field(default_factory=list)
    suspicious: list[dict] = field(default_factory=list)
    compliant: list[dict] = field(default_factory=list)
    files_scanned: int = 0

    def is_compliant(self) -> bool:
        """Check if repository is fully compliant"""
        return len(self.violations) == 0


class QuestionnaireArchitectureAuditor(ast.NodeVisitor):
    """AST visitor to detect questionnaire access violations"""

    def __init__(self, file_path: Path, source_code: str):
        self.file_path = file_path
        self.source_code = source_code
        self.source_lines = source_code.split('\n')
        self.violations: list[Violation] = []
        self.suspicious: list[dict] = []
        self.file_name = file_path.name
        self.relative_path = str(file_path.relative_to(REPO_ROOT))

        # Track imports
        self.imports_qrp = False
        self.has_from_file_call = False
        self.has_monolith_open = False

    def visit_Import(self, node: ast.Import) -> None:
        """Check for 'import questionnaire_resource_provider'"""
        for alias in node.names:
            if 'questionnaire_resource_provider' in alias.name:
                # Allow test files to import what they're testing
                if not is_test_file(self.relative_path) and self.file_name not in
ALLOWED_QRP_IMPORTERS:
                    self.violations.append(Violation(
                        file_path=self.relative_path,
                        line_number=node.lineno,
                        violation_type="ILLEGAL_IMPORT",
                        code_snippet=self._get_line(node.lineno),
                        explanation=f"Module {self.file_name} imports
questionnaire_resource_provider, "
                                    f"but only {ALLOWED_QRP_IMPORTERS} (and test files)
are allowed to import it."
                    ))
                self.imports_qrp = True
        self.generic_visit(node)

    def visit_ImportFrom(self, node: ast.ImportFrom) -> None:
        """Check for 'from questionnaire_resource_provider import ...'"""
        if node.module and 'questionnaire_resource_provider' in node.module:
            # Allow test files to import what they're testing
            if not is_test_file(self.relative_path) and self.file_name not in
ALLOWED_QRP_IMPORTERS:
                imported_names = ', '.join(alias.name for alias in node.names)
                self.violations.append(Violation(
                    file_path=self.relative_path,
                    line_number=node.lineno,
                    violation_type="ILLEGAL_IMPORT",
                    code_snippet=self._get_line(node.lineno),
```

```
                explanation=f"Module {self.file_name} imports {imported_names} from "
                            f"questionnaire_resource_provider, but only
{ALLOWED_QRP_IMPORTERS} "
                            f"(and test files) are allowed to import from it."
                ))
            self.imports_qrp = True
        self.generic_visit(node)

    def visit_Call(self, node: ast.Call) -> None:
        """Check for calls to QuestionnaireResourceProvider.from_file and file I/O"""
        # Check for QuestionnaireResourceProvider.from_file()
        if isinstance(node.func, ast.Attribute):
            if node.func.attr == 'from_file':
                if isinstance(node.func.value, ast.Name):
                    if node.func.value.id == 'QuestionnaireResourceProvider':
                        # Allow test files and factory to use from_file
                        if not is_test_file(self.relative_path) and self.file_name not in
ALLOWED_MONOLITH_READERS:
                            self.violations.append(Violation(
                                file_path=self.relative_path,
                                line_number=node.lineno,
                                violation_type="LEGACY_IO_PATH",
                                code_snippet=self._get_line(node.lineno),
                                explanation=f"Module {self.file_name} calls
QuestionnaireResourceProvider.from_file(), "
                                            f"which is a legacy I/O path. Only factory.py
(and tests) should use from_file(). "
                                            f"Core modules should receive QRP via
dependency injection."
                            ))
                        self.has_from_file_call = True

        # Check for open() calls with questionnaire_monolith
        if isinstance(node.func, ast.Name) and node.func.id == 'open':
            if node.args:
                arg = node.args[0]
                line_text = self._get_line(node.lineno)
                if 'questionnaire_monolith' in line_text.lower():
                    if self.file_name not in ALLOWED_MONOLITH_READERS:
                        self.violations.append(Violation(
                            file_path=self.relative_path,
                            line_number=node.lineno,
                            violation_type="ILLEGAL_DATA_ACCESS",
                            code_snippet=line_text,
                            explanation=f"Module {self.file_name} directly opens
questionnaire_monolith file. "
                                        f"Only factory.py is allowed to perform
questionnaire I/O."
                        ))
                    self.has_monolith_open = True

        # Check for json.load with questionnaire_monolith context
        if isinstance(node.func, ast.Attribute):
            if node.func.attr == 'load' and isinstance(node.func.value, ast.Name):
                if node.func.value.id == 'json':
                    line_text = self._get_line(node.lineno)
                    # Check context (look at surrounding lines)
                    context = self._get_context(node.lineno, 3)
                    if 'questionnaire_monolith' in context.lower():
                        if self.file_name not in ALLOWED_MONOLITH_READERS:
                            self.violations.append(Violation(
                                file_path=self.relative_path,
                                line_number=node.lineno,
                                violation_type="ILLEGAL_DATA_ACCESS",
                                code_snippet=line_text,
                                explanation=f"Module {self.file_name} loads JSON in
context of questionnaire_monolith. "
                                            f"Only factory.py is allowed to perform
```

```
questionnaire I/O."
                        ))

        self.generic_visit(node)

    def visit_FunctionDef(self, node: ast.FunctionDef) -> None:
        """Check for functions that look like pattern extraction or monolith loaders"""
        func_name = node.name.lower()

        # Check for functions that load from monolith
        if 'monolith' in func_name and ('load' in func_name or 'from' in func_name or
'read' in func_name):
            if self.file_name not in ALLOWED_MONOLITH_READERS:
                # Check if function actually reads files
                body_text = ast.get_source_segment(self.source_code, node) or ''
                if 'open(' in body_text or 'json.load' in body_text or 'read(' in
body_text:
                    self.violations.append(Violation(
                        file_path=self.relative_path,
                        line_number=node.lineno,
                        violation_type="ILLEGAL_DATA_ACCESS",
                        code_snippet=f"def {node.name}(...)",
                        explanation=f"Function {node.name} loads questionnaire monolith
data. "
                                    f"Only factory.py is allowed to perform questionnaire
I/O. "
                                    f"Use factory.load_questionnaire_monolith() instead."
                    ))

        # Pattern extraction indicators
        pattern_keywords = ['extract_pattern', 'derive_pattern', 'build_pattern',
'compile_pattern',
                            'get_pattern', 'pattern_from', 'validation_from',
'extract_validation']

        if any(keyword in func_name for keyword in pattern_keywords):
            if self.file_name not in ['questionnaire_resource_provider.py', 'factory.py']:
                # Check if function body contains questionnaire-related logic
                body_text = ast.get_source_segment(self.source_code, node) or ''
                if any(word in body_text.lower() for word in ['questionnaire', 'monolith',
 'pattern', 'validation']):
                    self.suspicious.append({
                        'file': self.relative_path,
                        'line': node.lineno,
                        'function': node.name,
                        'reason': f"Function name suggests pattern extraction logic. "
                                  f"Pattern logic should only exist in
QuestionnaireResourceProvider."
                    })

        self.generic_visit(node)

    def visit_ClassDef(self, node: ast.ClassDef) -> None:
        """Check for classes that look like pattern providers"""
        class_name = node.name.lower()

        if 'pattern' in class_name or 'questionnaire' in class_name or 'validation' in
class_name:
            if self.file_name not in ['questionnaire_resource_provider.py', 'factory.py',
'contracts.py']:
                self.suspicious.append({
                    'file': self.relative_path,
                    'line': node.lineno,
                    'class': node.name,
                    'reason': f"Class name suggests questionnaire/pattern logic. "
                              f"Such logic should only exist in
QuestionnaireResourceProvider."
                })
```

```
        self.generic_visit(node)

    def _get_line(self, line_num: int) -> str:
        """Get source line at given line number"""
        if 1 <= line_num <= len(self.source_lines):
            return self.source_lines[line_num - 1].strip()
        return ""

    def _get_context(self, line_num: int, lines_before: int = 2, lines_after: int = 2) ->
str:
        """Get context around a line"""
        start = max(1, line_num - lines_before)
        end = min(len(self.source_lines), line_num + lines_after)
        return '\n'.join(self.source_lines[start-1:end])


def scan_file(file_path: Path) -> tuple[list[Violation], list[dict]]:
    """Scan a single Python file for violations"""
    try:
        source = file_path.read_text(encoding='utf-8')
        tree = ast.parse(source, filename=str(file_path))

        auditor = QuestionnaireArchitectureAuditor(file_path, source)
        auditor.visit(tree)

        return auditor.violations, auditor.suspicious
    except SyntaxError as e:
        print(f"Syntax error in {file_path}: {e}", file=sys.stderr)
        return [], []
    except Exception as e:
        print(f"Error scanning {file_path}: {e}", file=sys.stderr)
        return [], []


def scan_repository() -> AnalysisReport:
    """Scan entire repository for violations"""
    report = AnalysisReport()

    # Find all Python files
    python_files = list(REPO_ROOT.rglob('*.py'))

    # Exclude test files, migrations, and virtual environments
    python_files = [
        f for f in python_files
        if not any(part.startswith('.') or part in ['venv', 'env', '__pycache__',
'migrations']
                   for part in f.parts)
    ]

    print(f"Scanning {len(python_files)} Python files...")

    for file_path in python_files:
        violations, suspicious = scan_file(file_path)
        report.violations.extend(violations)
        report.suspicious.extend(suspicious)
        report.files_scanned += 1

        # Track compliant files that correctly use the architecture
        if not violations and not suspicious:
            # Check if file uses QRP correctly (via factory)
            try:
                source = file_path.read_text(encoding='utf-8')
                if 'QuestionnaireResourceProvider' in source or
'questionnaire_resource_provider' in source:
                    relative_path = str(file_path.relative_to(REPO_ROOT))
                    if file_path.name in ALLOWED_QRP_IMPORTERS:
                        report.compliant.append({
```

```
                    'file': relative_path,
                    'reason': 'Correctly imports/uses
QuestionnaireResourceProvider per architecture'
                })
        except:
            pass

    return report


def generate_report(report: AnalysisReport) -> str:
    """Generate comprehensive compliance report"""
    lines = []

    # Compliance Summary
    lines.append("=" * 80)
    lines.append("QUESTIONNAIRE ARCHITECTURE COMPLIANCE REPORT")
    lines.append("=" * 80)
    lines.append("")

    lines.append("COMPLIANCE SUMMARY")
    lines.append("-" * 80)

    if report.is_compliant():
        lines.append("✓ COMPLIANT: The repository fully adheres to the Questionnaire
Access Architecture.")
        lines.append(f"  Scanned {report.files_scanned} files with NO violations
detected.")
    else:
        lines.append(f"✗ NON-COMPLIANT: {len(report.violations)} architectural violations
detected.")
        lines.append(f"  The repository violates the Questionnaire Access Architecture
specification.")
        lines.append(f"  Scanned {report.files_scanned} files.")

    lines.append("")

    # Violations
    if report.violations:
        lines.append("VIOLATIONS")
        lines.append("-" * 80)
        lines.append("")

        # Group by type
        by_type: dict[str, list[Violation]] = {}
        for v in report.violations:
            by_type.setdefault(v.violation_type, []).append(v)

        for vtype, violations in sorted(by_type.items()):
            lines.append(f"[{vtype}] ({len(violations)} occurrence(s))")
            lines.append("")

            for i, v in enumerate(violations, 1):
                lines.append(f"  {i}. File: {v.file_path}")
                lines.append(f"     Line: {v.line_number}")
                lines.append(f"     Code: {v.code_snippet}")
                lines.append(f"     Explanation: {v.explanation}")
                lines.append("")

        lines.append("")

    # Suspicious Constructs
    if report.suspicious:
        lines.append("SUSPICIOUS CONSTRUCTS")
        lines.append("-" * 80)
        lines.append("")

        for i, sus in enumerate(report.suspicious, 1):
```

```
            lines.append(f"  {i}. File: {sus['file']}")
            lines.append(f"     Line: {sus['line']}")
            if 'function' in sus:
                lines.append(f"     Function: {sus['function']}")
            if 'class' in sus:
                lines.append(f"     Class: {sus['class']}")
            lines.append(f"     Reason: {sus['reason']}")
            lines.append("")

        lines.append("")

    # Compliant Access Points
    if report.compliant:
        lines.append("CONFIRMED COMPLIANT ACCESS POINTS")
        lines.append("-" * 80)
        lines.append("")

        for comp in report.compliant:
            lines.append(f"  ✓ {comp['file']}")
            lines.append(f"    {comp['reason']}")
            lines.append("")

        lines.append("")

    # Remediation Guidance
    if report.violations:
        lines.append("REMEDIATION GUIDANCE")
        lines.append("-" * 80)
        lines.append("")

        violation_types = set(v.violation_type for v in report.violations)

        if 'ILLEGAL_IMPORT' in violation_types:
            lines.append("ILLEGAL_IMPORT:")
            lines.append("  - Remove all imports of questionnaire_resource_provider from
modules outside")
            lines.append("    the allowed set: {factory.py, core.py,
questionnaire_resource_provider.py}")
            lines.append("  - Instead, receive QuestionnaireResourceProvider via
dependency injection")
            lines.append("  - Use factory.build_processor() to get a properly wired
ProcessorBundle")
            lines.append("")

        if 'ILLEGAL_DATA_ACCESS' in violation_types:
            lines.append("ILLEGAL_DATA_ACCESS:")
            lines.append("  - Remove all direct file I/O to questionnaire_monolith.json")
            lines.append("  - Use factory.load_questionnaire_monolith() for I/O-based
initialization")
            lines.append("  - Use factory.build_processor() to get pre-loaded
questionnaire data")
            lines.append("  - Pass questionnaire data via contracts/parameters, not by
reading files")
            lines.append("")

        if 'LEGACY_IO_PATH' in violation_types:
            lines.append("LEGACY_IO_PATH:")
            lines.append("  - Replace QuestionnaireResourceProvider.from_file() calls with
 factory-based initialization")
            lines.append("  - In factory.py: use load_questionnaire_monolith() then
construct QRP with that data")
            lines.append("  - In core modules: receive QRP via dependency injection, don't
 construct it")
            lines.append("  - Mark from_file() as @deprecated with migration guidance")
            lines.append("")

        if 'REIMPLEMENTED_QUESTIONNAIRE_LOGIC' in violation_types:
            lines.append("REIMPLEMENTED_QUESTIONNAIRE_LOGIC:")
```

```
        lines.append("  - Move all pattern extraction logic into
questionnaire_resource_provider.py")
        lines.append("  - Expose pattern catalogs via QuestionnaireResourceProvider
methods")
        lines.append("  - Remove duplicate pattern definitions from other modules")
        lines.append("  - Use provider.get_temporal_patterns(),
provider.get_indicator_patterns(), etc.")
        lines.append("")

    return '\n'.join(lines)


def main():
    """Run the audit and generate report"""
    print("Starting Questionnaire Architecture Enforcement Audit...")
    print()

    report = scan_repository()

    # Generate text report
    text_report = generate_report(report)
    print(text_report)

    # Save to file
    output_file = REPO_ROOT / 'ARCHITECTURE_AUDIT_REPORT.txt'
    output_file.write_text(text_report, encoding='utf-8')
    print(f"\nReport saved to: {output_file}")

    # Save JSON report for programmatic processing
    json_report = {
        'compliant': report.is_compliant(),
        'files_scanned': report.files_scanned,
        'violations': [
            {
                'file': v.file_path,
                'line': v.line_number,
                'type': v.violation_type,
                'code': v.code_snippet,
                'explanation': v.explanation
            }
            for v in report.violations
        ],
        'suspicious': report.suspicious,
        'compliant_files': report.compliant
    }

    json_file = REPO_ROOT / 'ARCHITECTURE_AUDIT_REPORT.json'
    json_file.write_text(json.dumps(json_report, indent=2), encoding='utf-8')
    print(f"JSON report saved to: {json_file}")

    # Exit with appropriate code
    sys.exit(0 if report.is_compliant() else 1)


if __name__ == '__main__':
    main()

===== FILE: scripts/audit_catalog_registry_alignment.py =====
#!/usr/bin/env python3
"""
Catalog-Registry-Usage Alignment Audit

Comprehensive audit to verify alignment between:
1. canonical_method_catalog.json (canonical method universe - 1,996 methods)
2. calibration_registry.py (calibration metadata)
3. Actual codebase usage

Outputs:
```

```
- Methods catalogued vs used vs calibrated
- Defects found
- Alignment verification results
"""

import json
import sys
from pathlib import Path
from collections import defaultdict

# Add src to path
repo_root = Path(__file__).parent.parent

from saaaaaa.core.orchestrator.calibration_registry import CALIBRATIONS


def main():
    print("="*80)
    print("CATALOG-REGISTRY-USAGE ALIGNMENT AUDIT")
    print("="*80)

    # Load canonical method catalog
    catalog_path = repo_root / "config" / "canonical_method_catalog.json"
    with open(catalog_path) as f:
        catalog_data = json.load(f)

    print(f"\nCanonical catalog: {catalog_data['summary']['total_methods']} methods
(v{catalog_data['metadata']['version']})")

    # Load usage intelligence
    usage_path = repo_root / "config" / "method_usage_intelligence.json"
    with open(usage_path, 'r') as f:
        usage_data = json.load(f)

    # Load calibration decisions
    decisions_path = repo_root / "config" / "calibration_decisions.json"
    with open(decisions_path, 'r') as f:
        decisions_data = json.load(f)

    # Build method sets
    catalog_methods = {
        (m['class_name'], m['method_name'])
        for m in catalog_data['methods']
        if m['class_name']
    }
    registry_methods = set(CALIBRATIONS.keys())
    used_methods = set()

    for fqn, usage in usage_data.get('methods', {}).items():
        if isinstance(usage, dict):
            class_name = usage.get('class_name', '')
            method_name = usage.get('method_name', '')
            if class_name and method_name and usage.get('total_usages', 0) > 0:
                used_methods.add((class_name, method_name))

    print(f"\n[INVENTORY]")
    print(f"  Catalog methods: {len(catalog_methods)}")
    print(f"  Registry methods: {len(registry_methods)}")
    print(f"  Used methods (in codebase): {len(used_methods)}")

    # Analyze overlaps
    print(f"\n[ALIGNMENT ANALYSIS]")

    # 1. Methods in catalog AND registry
    catalog_and_registry = catalog_methods & registry_methods
    print(f"  ✓ In both catalog AND registry: {len(catalog_and_registry)}")

    # 2. Methods in catalog but NOT in registry
```

```
    catalog_not_registry = catalog_methods - registry_methods
    print(f"  ⚠ In catalog but NOT in registry: {len(catalog_not_registry)}")

    # 3. Methods in registry but NOT in catalog (DEFECT)
    registry_not_catalog = registry_methods - catalog_methods
    print(f"  ✗ In registry but NOT in catalog (DEFECT): {len(registry_not_catalog)}")

    # 4. Methods used but NOT in catalog (DEFECT)
    used_not_catalog = used_methods - catalog_methods
    print(f"  ✗ Used but NOT in catalog (DEFECT): {len(used_not_catalog)}")

    # 5. Methods in catalog but NEVER used
    catalog_not_used = catalog_methods - used_methods
    print(f"  ⚠ In catalog but NEVER used: {len(catalog_not_used)}")

    # 6. Methods used but NOT in registry
    used_not_registry = used_methods - registry_methods
    print(f"  ⚠ Used but NOT in registry: {len(used_not_registry)}")

    # Build defect report
    defects = []
    acceptable_divergence = []

    # Check 1: Registry methods not in catalog
    # Per CATALOG_REGISTRY_ALIGNMENT_POLICY.md, this is ACCEPTABLE if methods are unused
    for class_name, method_name in sorted(registry_not_catalog):
        fqn = f"{class_name}.{method_name}"

        # Check if this method is actually used
        usage = usage_data.get('methods', {}).get(fqn, {})
        usage_count = usage.get('total_usages', 0) if isinstance(usage, dict) else 0

        if usage_count > 0:
            # Used but not in catalog - this is a DEFECT
            defects.append({
                "type": "REGISTRY_NOT_IN_CATALOG_USED",
                "severity": "HIGH",
                "method": fqn,
                "description": f"Method has calibration, is USED ({usage_count} times),
but not in canonical catalog",
                "action": "Add to catalog - this method is actively used"
            })
        else:
            # Unused and not in catalog - this is ACCEPTABLE per policy
            acceptable_divergence.append({
                "type": "REGISTRY_NOT_IN_CATALOG_UNUSED",
                "severity": "INFO",
                "method": fqn,
                "description": "Method has calibration but is not in catalog (unused -
acceptable per policy)",
                "action": "No action required (different scopes) - see
CATALOG_REGISTRY_ALIGNMENT_POLICY.md"
            })

    # Check 2: Used method not in catalog - ALWAYS A DEFECT
    for class_name, method_name in sorted(used_not_catalog):
        defects.append({
            "type": "USED_NOT_IN_CATALOG",
            "severity": "CRITICAL",
            "method": f"{class_name}.{method_name}",
            "description": "Method is used in codebase but not in canonical catalog",
            "action": "Add to catalog immediately"
        })

    # Warnings
    warnings = []

    # Warning Type 1: Catalog method never used
```

```
    for class_name, method_name in sorted(list(catalog_not_used)[:20]):  # Top 20
        warnings.append({
            "type": "CATALOGUED_NOT_USED",
            "severity": "LOW",
            "method": f"{class_name}.{method_name}",
            "description": "Method in catalog but never used in codebase",
            "action": "Consider if method is obsolete"
        })

    # Warning Type 2: Used but not calibrated
    for class_name, method_name in sorted(list(used_not_registry)[:20]):  # Top 20
        # Check calibration decision
        fqn = f"{class_name}.{method_name}"

        # Decisions are now method-keyed, not category-keyed
        decision_data = decisions_data.get('decisions', {}).get(fqn)

        if decision_data and decision_data.get('decision') == "REQUIRES_CALIBRATION":
            warnings.append({
                "type": "USED_NOT_CALIBRATED",
                "severity": "MEDIUM",
                "method": fqn,
                "description": "Method is used and requires calibration but not in
registry",
                "action": f"Add calibration entry (auto-decision:
{decision_data.get('decision')})"
            })

    # Generate report
    report = {
        "metadata": {
            "generated_at": "2025-11-08",
            "audit_version": "1.0.0"
        },
        "inventory": {
            "catalog_methods": len(catalog_methods),
            "registry_methods": len(registry_methods),
            "used_methods": len(used_methods),
        },
        "alignment": {
            "catalog_and_registry": len(catalog_and_registry),
            "catalog_not_registry": len(catalog_not_registry),
            "registry_not_catalog": len(registry_not_catalog),
            "used_not_catalog": len(used_not_catalog),
            "catalog_not_used": len(catalog_not_used),
            "used_not_registry": len(used_not_registry),
        },
        "defects": defects,
        "acceptable_divergence": acceptable_divergence,
        "warnings": warnings[:50],  # Limit warnings
        "alignment_score": {
            "catalog_registry_alignment": round(len(catalog_and_registry) /
max(len(catalog_methods), 1) * 100, 2),
            "catalog_usage_alignment": round(len(catalog_methods - catalog_not_used) /
max(len(catalog_methods), 1) * 100, 2),
            "overall_integrity": "PASS" if len(defects) == 0 else "FAIL"
        }
    }

    # Write report
    output_path = repo_root / "config" / "alignment_audit_report.json"
    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(report, f, indent=2, ensure_ascii=False)

    print(f"\n\n[DEFECT REPORT]")
    print(f"  Total CRITICAL defects: {len(defects)}")

    if defects:
```

```
        print(f"\n  CRITICAL defects (require action):")
        for defect in defects[:10]:
            print(f"    {defect['type']}: {defect['method']}")
            print(f"      → {defect['description']}")
            print(f"      Action: {defect['action']}")

    print(f"\n\n[ACCEPTABLE DIVERGENCE]")
    print(f"  Total acceptable divergences: {len(acceptable_divergence)}")
    print(f"  (Registry methods not in catalog but unused - per
CATALOG_REGISTRY_ALIGNMENT_POLICY.md)")

    if acceptable_divergence and len(acceptable_divergence) <= 10:
        print(f"\n  All {len(acceptable_divergence)} acceptable divergences:")
        for item in acceptable_divergence:
            print(f"    {item['method']}")
    elif acceptable_divergence:
        print(f"\n  Sample acceptable divergences (first 5 of
{len(acceptable_divergence)}):")
        for item in acceptable_divergence[:5]:
            print(f"    {item['method']}")
        print(f"  See CATALOG_REGISTRY_ALIGNMENT_POLICY.md for full policy")

    print(f"\n\n[WARNING REPORT]")
    print(f"  Total warnings: {len(warnings)}")

    if warnings:
        print(f"\n  Sample warnings (first 5):")
        for warning in warnings[:5]:
            print(f"    {warning['type']}: {warning['method']}")
            print(f"      → {warning['description']}")

    print(f"\n\n[ALIGNMENT SCORES]")
    print(f"  Catalog-Registry alignment:
{report['alignment_score']['catalog_registry_alignment']}%")
    print(f"  Catalog-Usage alignment:
{report['alignment_score']['catalog_usage_alignment']}%")
    print(f"  Overall integrity: {report['alignment_score']['overall_integrity']}")

    print(f"\n✓ Audit report written to: {output_path}")

    # Policy-aware exit
    if len(defects) > 0:
        print(f"\n✗ AUDIT FAILED: {len(defects)} CRITICAL defects found")
        print("   Fix critical defects before proceeding")
        print(f"\n   Note: {len(acceptable_divergence)} acceptable divergences
documented")
        print("   See CATALOG_REGISTRY_ALIGNMENT_POLICY.md for alignment policy")
        return 1
    else:
        print(f"\n✓ AUDIT PASSED: No critical defects found")
        print(f"   ({len(acceptable_divergence)} acceptable divergences per policy)")
        return 0


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

===== FILE: scripts/audit_circular_imports.py =====
#!/usr/bin/env python3
"""
Circular Import Detector

Detects circular import patterns in the codebase using AST analysis.
Circular imports can cause import failures and are difficult to debug.

This script builds an import graph and detects cycles.

Exit codes:
```

```
- 0: No circular imports detected
- 1: Circular imports detected (with details)
"""

from __future__ import annotations

import ast
import sys
from collections import defaultdict
from pathlib import Path
from typing import Set


def extract_imports(file_path: Path) -> set[str]:
    """
    Extract all imported modules from a Python file.

    Returns
    -------
    set[str]
        Set of imported module names (absolute imports only)
    """
    try:
        with open(file_path, "r", encoding="utf-8") as f:
            tree = ast.parse(f.read(), filename=str(file_path))
    except (SyntaxError, UnicodeDecodeError):
        return set()

    imports = set()

    for node in ast.walk(tree):
        if isinstance(node, ast.Import):
            for alias in node.names:
                # Keep full module path for better cycle detection
                imports.add(alias.name)

        elif isinstance(node, ast.ImportFrom):
            if node.module and node.level == 0:  # Absolute import
                # Keep full module path for better cycle detection
                imports.add(node.module)

    return imports


def build_import_graph(root: Path, package_name: str = "saaaaaa") -> dict[str, set[str]]:
    """
    Build import graph for the package.

    Parameters
    ----------
    root : Path
        Root directory of the project
    package_name : str
        Name of the package to analyze

    Returns
    -------
    dict[str, set[str]]
        Adjacency list representing import dependencies
    """
    graph = defaultdict(set)
    src_root = root / "src" / package_name

    if not src_root.exists():
        print(f"Warning: Package directory not found: {src_root}", file=sys.stderr)
        return dict(graph)

    # Find all Python files in the package
```

```
    python_files = list(src_root.rglob("*.py"))

    # Build module name to file path mapping
    file_to_module = {}
    for py_file in python_files:
        rel_path = py_file.relative_to(src_root)
        parts = list(rel_path.parts)

        # Convert to module name
        if parts[-1] == "__init__.py":
            parts = parts[:-1]
        else:
            parts[-1] = parts[-1].replace(".py", "")

        if parts:
            module_name = ".".join(parts)
            file_to_module[py_file] = f"{package_name}.{module_name}"
        else:
            file_to_module[py_file] = package_name

    # Build dependency graph
    for py_file, module_name in file_to_module.items():
        imports = extract_imports(py_file)

        # Filter to only imports within our package
        for imp in imports:
            if imp == package_name or imp.startswith(f"{package_name}."):
                graph[module_name].add(imp)

    return dict(graph)


def find_cycles(graph: dict[str, set[str]]) -> list[list[str]]:
    """
    Find all cycles in the import graph using DFS.

    Parameters
    ----------
    graph : dict[str, set[str]]
        Import dependency graph

    Returns
    -------
    list[list[str]]
        List of cycles (each cycle is a list of module names)
    """
    cycles = []
    visited = set()
    rec_stack = set()
    path = []

    def dfs(node: str) -> None:
        visited.add(node)
        rec_stack.add(node)
        path.append(node)

        for neighbor in graph.get(node, set()):
            if neighbor not in visited:
                dfs(neighbor)
            elif neighbor in rec_stack:
                # Found a cycle
                cycle_start = path.index(neighbor)
                cycle = path[cycle_start:] + [neighbor]
                cycles.append(cycle)

        path.pop()
        rec_stack.remove(node)
```

```
    for node in graph:
        if node not in visited:
            dfs(node)

    return cycles


def main() -> int:
    """Main entry point."""
    root = Path(__file__).parent.parent

    print("=== Circular Import Detection ===")
    print(f"Scanning: {root}")
    print()

    # Build import graph
    print("Building import graph...")
    graph = build_import_graph(root)
    print(f"Found {len(graph)} modules with imports")
    print()

    # Detect cycles
    print("Detecting circular imports...")
    cycles = find_cycles(graph)

    if not cycles:
        print("✓ No circular imports detected")
        return 0

    print(f"✗ Found {len(cycles)} circular import(s):\n")

    for i, cycle in enumerate(cycles, 1):
        print(f"Cycle {i}:")
        for j, module in enumerate(cycle[:-1]):
            print(f"  {module}")
            if j < len(cycle) - 2:
                print(f"    ↓ imports")
        print()

    print(f"Total cycles: {len(cycles)}")
    print("\nCircular imports can cause import failures and runtime errors.")
    print("Fixes:")
    print("  1. Move import inside function (deferred import)")
    print("  2. Refactor to break dependency")
    print("  3. Introduce intermediate module")

    return 1


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

===== FILE: scripts/audit_dependencies.py =====
#!/usr/bin/env python3
"""
Comprehensive Dependency Auditor for SAAAAAA Project

This script performs exhaustive dependency analysis:
1. Static AST analysis to extract all imports
2. Classification by role (core_runtime, optional_runtime, dev_test, docs)
3. Detection of missing dependencies
4. Version pinning recommendations
5. Generation of structured dependency files
"""

import ast
import importlib
import importlib.metadata
```

```
import json
import sys
from collections import defaultdict
from pathlib import Path
from typing import Dict, List, Set, Tuple

# Mapping of import names to PyPI package names
IMPORT_TO_PACKAGE = {
    "sklearn": "scikit-learn",
    "cv2": "opencv-python",
    "PIL": "Pillow",
    "yaml": "pyyaml",
    "dotenv": "python-dotenv",
    "jwt": "pyjwt",
    "socketio": "python-socketio",
    "blake3": "blake3",
    "bs4": "beautifulsoup4",
    "OpenSSL": "pyOpenSSL",
    "MySQLdb": "mysqlclient",
    "psycopg2": "psycopg2-binary",
    "fitz": "PyMuPDF",
    "docx": "python-docx",
    "flask_socketio": "flask-socketio",
    "flask_cors": "flask-cors",
    "sse_starlette": "sse-starlette",
    "tabula": "tabula-py",
    "camelot": "camelot-py",
}

# Local project modules that should not be treated as external dependencies
LOCAL_MODULES = {
    "saaaaaa",
    # Project internal modules
    "advanced_module_config", "config", "layer_requirements", "tomllib",
    "calibration_context", "intrinsic_loader", "event_tracking",
    "layer_coexistence", "congruence_layer", "saga", "compatibility",
    "executor_config", "data_structures", "calibration_registry",
    "analysis", "errors", "scripts", "feature_flags", "unit_layer",
    "signal_consumption", "questionnaire", "meta_layer", "pdt_structure",
    "enhanced_contracts", "audit_system", "contract_io", "processing",
    "opentelemetry_integration", "src", "core_module_factory", "lazy_deps",
    "chain_layer", "layer_computers", "safe_imports", "versions",
    "utils", "dependency_lockdown", "choquet_aggregator",
    "questionnaire_resource_provider", "signals", "rl_strategy",
    "signal_loader", "policy_processor", "golden_rule", "scoring",
    "validation", "financiero_viabilidad_tablas", "aggregation_models",
    "evidence_registry", "tables", "schemas", "document_ingestion",
    "seeds", "core", "chunking", "parsers", "retry_handler",
    "inference", "dnp_integration", "class_registry", "factory",
    "concurrency", "contract_loader", "phases", "schema_validator",
    "tests", "contradiction_deteccion", "contracts", "methods",
    "structural", "runtime_error_fixes", "tools",
    "architecture_validator", "arg_router", "quality_gates", "pipeline",
    "configs", "cli", "models", "recommendation_engine",
    # Executors
    "executors",
    # Orchestrator modules
    "orchestrator",
    # API modules
    "api",
}

# Standard library modules (Python 3.10+) - should not be listed as dependencies
# Use sys.stdlib_module_names for Python 3.10+, with fallback for older versions
import sys as _sys

if hasattr(_sys, 'stdlib_module_names'):
    STDLIB_MODULES = _sys.stdlib_module_names
```

```
else:
    # Fallback for Python <3.10 - manually maintained list
    STDLIB_MODULES = {
        "abc", "argparse", "ast", "asyncio", "base64", "collections", "concurrent",
        "contextlib", "copy", "dataclasses", "datetime", "decimal", "enum", "functools",
        "hashlib", "heapq", "importlib", "inspect", "io", "itertools", "json", "logging",
        "math", "multiprocessing", "operator", "os", "pathlib", "pickle", "platform",
        "queue", "random", "re", "shutil", "signal", "socket", "sqlite3", "statistics",
        "string", "struct", "subprocess", "sys", "tempfile", "textwrap", "threading",
        "time", "traceback", "types", "typing", "unittest", "urllib", "uuid", "warnings",
        "weakref", "xml", "zipfile", "zoneinfo"
    }

# Package role classifications
ROLE_CLASSIFICATIONS = {
    # Core runtime - critical for production execution
    "core_runtime": {
        "numpy", "pandas", "polars", "pyarrow", "scipy", "scikit-learn",
        "torch", "tensorflow", "transformers", "sentence-transformers",
        "spacy", "networkx", "pymc", "arviz", "pytensor",
        "pdfplumber", "PyPDF2", "PyMuPDF", "python-docx",
        "flask", "fastapi", "httpx", "uvicorn", "sse-starlette",
        "pydantic", "pyyaml", "jsonschema", "blake3",
        "structlog", "opentelemetry-api", "opentelemetry-sdk",
        "tenacity", "typer", "python-dotenv"
    },
    # Optional runtime - enhances functionality but not critical
    "optional_runtime": {
        "flask-cors", "flask-socketio", "python-socketio",
        "gevent", "gevent-websocket", "pyjwt",
        "redis", "sqlalchemy", "gunicorn",
        "prometheus-client", "psutil",
        "opentelemetry-instrumentation-fastapi",
        "dowhy", "econml", "igraph", "python-louvain", "pydot",
        "tabula-py", "camelot-py",
        "nltk", "sentencepiece", "tiktoken", "fuzzywuzzy",
        "python-Levenshtein", "langdetect"
    },
    # Development & testing
    "dev_test": {
        "pytest", "pytest-cov", "hypothesis", "schemathesis",
        "black", "ruff", "flake8", "mypy", "pyright",
        "bandit", "pycycle", "import-linter"
    },
    # Documentation
    "docs": {
        "sphinx", "sphinx-rtd-theme", "myst-parser"
    }
}


class DependencyAuditor:
    """Audits and classifies all dependencies in the project."""

    def __init__(self, project_root: Path):
        self.project_root = project_root
        self.imports_by_file: Dict[str, Set[str]] = defaultdict(set)
        self.all_imports: Set[str] = set()
        self.missing_packages: Set[str] = set()
        self.installed_packages: Dict[str, str] = {}
        self.package_usage: Dict[str, List[str]] = defaultdict(list)

    def scan_imports_in_file(self, filepath: Path) -> Set[str]:
        """Extract all imports from a Python file using AST."""
        imports = set()
        try:
            with open(filepath, 'r', encoding='utf-8') as f:
                tree = ast.parse(f.read(), filename=str(filepath))
```

```
            for node in ast.walk(tree):
                if isinstance(node, ast.Import):
                    for alias in node.names:
                        imports.add(alias.name.split('.')[0])
                elif isinstance(node, ast.ImportFrom):
                    if node.module:
                        imports.add(node.module.split('.')[0])
        except Exception as e:
            print(f"Warning: Could not parse {filepath}: {e}", file=sys.stderr)

        return imports

    def scan_all_python_files(self):
        """Scan all Python files in the project."""
        print("Scanning Python files for imports...")

        # Scan src directory
        src_dir = self.project_root / "src"
        if src_dir.exists():
            for py_file in src_dir.rglob("*.py"):
                if "__pycache__" not in str(py_file):
                    rel_path = py_file.relative_to(self.project_root)
                    imports = self.scan_imports_in_file(py_file)
                    self.imports_by_file[str(rel_path)] = imports
                    self.all_imports.update(imports)

        # Scan root level Python files
        for py_file in self.project_root.glob("*.py"):
            if py_file.name not in ["setup.py"]:
                rel_path = py_file.relative_to(self.project_root)
                imports = self.scan_imports_in_file(py_file)
                self.imports_by_file[str(rel_path)] = imports
                self.all_imports.update(imports)

        # Scan tests directory
        tests_dir = self.project_root / "tests"
        if tests_dir.exists():
            for py_file in tests_dir.rglob("*.py"):
                if "__pycache__" not in str(py_file):
                    rel_path = py_file.relative_to(self.project_root)
                    imports = self.scan_imports_in_file(py_file)
                    self.imports_by_file[str(rel_path)] = imports
                    # Mark test imports separately
                    self.all_imports.update(imports)

        # Scan examples directory
        examples_dir = self.project_root / "examples"
        if examples_dir.exists():
            for py_file in examples_dir.rglob("*.py"):
                if "__pycache__" not in str(py_file):
                    rel_path = py_file.relative_to(self.project_root)
                    imports = self.scan_imports_in_file(py_file)
                    self.imports_by_file[str(rel_path)] = imports
                    self.all_imports.update(imports)

        # Scan scripts directory
        scripts_dir = self.project_root / "scripts"
        if scripts_dir.exists():
            for py_file in scripts_dir.rglob("*.py"):
                if "__pycache__" not in str(py_file):
                    rel_path = py_file.relative_to(self.project_root)
                    imports = self.scan_imports_in_file(py_file)
                    self.imports_by_file[str(rel_path)] = imports
                    self.all_imports.update(imports)

        print(f"Found {len(self.imports_by_file)} Python files")
        print(f"Found {len(self.all_imports)} unique imports")
```

```
def get_installed_packages(self):
    """Get all currently installed packages and their versions."""
    print("Checking installed packages...")
    try:
        for dist in importlib.metadata.distributions():
            self.installed_packages[dist.name.lower()] = dist.version
    except Exception as e:
        print(f"Warning: Could not get installed packages: {e}", file=sys.stderr)

def normalize_import_to_package(self, import_name: str) -> str:
    """Convert import name to PyPI package name."""
    # Check if it's a known mapping
    if import_name in IMPORT_TO_PACKAGE:
        return IMPORT_TO_PACKAGE[import_name]

    # Check if it's a local package
    if import_name == "saaaaaa":
        return "saaaaaa"

    # Otherwise, assume import name = package name
    return import_name

def classify_import(self, import_name: str) -> str:
    """Classify an import into a role category."""
    # Skip stdlib modules
    if import_name in STDLIB_MODULES:
        return "stdlib"

    # Skip local modules
    if import_name in LOCAL_MODULES:
        return "local"

    package_name = self.normalize_import_to_package(import_name)

    # Check role classifications
    for role, packages in ROLE_CLASSIFICATIONS.items():
        if package_name in packages:
            return role

    # Default: assume core_runtime for unknown packages
    return "core_runtime"

def check_importability(self, import_name: str) -> Tuple[bool, str]:
    """Check if a module can be imported."""
    if import_name in STDLIB_MODULES:
        return True, "stdlib"

    if import_name in LOCAL_MODULES:
        return True, "local"

    try:
        importlib.import_module(import_name)
        return True, "available"
    except ImportError:
        return False, "missing"
    except Exception as e:
        return False, f"error: {str(e)[:50]}"

def build_package_usage_map(self):
    """Build a map of which files use which packages."""
    print("Building package usage map...")

    for filepath, imports in self.imports_by_file.items():
        for import_name in imports:
            if import_name not in STDLIB_MODULES and import_name not in LOCAL_MODULES:
                package_name = self.normalize_import_to_package(import_name)
                self.package_usage[package_name].append(filepath)
```

```
    def detect_missing_dependencies(self):
        """Detect which imports cannot be satisfied."""
        print("Detecting missing dependencies...")

        for import_name in self.all_imports:
            if import_name in STDLIB_MODULES or import_name in LOCAL_MODULES:
                continue

            can_import, status = self.check_importability(import_name)
            if not can_import:
                package_name = self.normalize_import_to_package(import_name)
                self.missing_packages.add(package_name)
                print(f"  Missing: {import_name} -> {package_name} ({status})")

    def generate_report(self) -> Dict:
        """Generate comprehensive audit report."""
        report = {
            "summary": {
                "total_files_scanned": len(self.imports_by_file),
                "total_unique_imports": len(self.all_imports),
                "missing_packages": len(self.missing_packages),
                "installed_packages": len(self.installed_packages)
            },
            "imports_by_file": {},
            "package_classification": defaultdict(list),
            "missing_packages": list(self.missing_packages),
            "package_usage": {}
        }

        # Classify all non-stdlib, non-local imports
        for import_name in sorted(self.all_imports):
            if import_name not in STDLIB_MODULES and import_name not in LOCAL_MODULES:
                role = self.classify_import(import_name)
                package_name = self.normalize_import_to_package(import_name)
                if package_name not in report["package_classification"][role]:
                    report["package_classification"][role].append(package_name)

        # Add package usage
        for package, files in sorted(self.package_usage.items()):
            report["package_usage"][package] = files

        return report

    def run_full_audit(self) -> Dict:
        """Run complete dependency audit."""
        print("=" * 80)
        print("DEPENDENCY AUDIT STARTING")
        print("=" * 80)

        self.scan_all_python_files()
        self.get_installed_packages()
        self.build_package_usage_map()
        self.detect_missing_dependencies()

        report = self.generate_report()

        print("\n" + "=" * 80)
        print("AUDIT COMPLETE")
        print("=" * 80)
        print(f"Files scanned: {report['summary']['total_files_scanned']}")
        print(f"Unique imports: {report['summary']['total_unique_imports']}")
        print(f"Missing packages: {report['summary']['missing_packages']}")

        return report


def main():
```

```
    """Main entry point."""
    project_root = Path(__file__).parent.parent
    auditor = DependencyAuditor(project_root)

    report = auditor.run_full_audit()

    # Save report to JSON
    output_file = project_root / "dependency_audit_report.json"
    with open(output_file, 'w', encoding='utf-8') as f:
        json.dump(report, f, indent=2, sort_keys=True, default=list)

    print(f"\nReport saved to: {output_file}")

    # Return non-zero exit code if missing packages
    if report['summary']['missing_packages'] > 0:
        print("\n⚠  WARNING: Missing packages detected!")
        return 1

    return 0


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

===== FILE: scripts/audit_import_budget.py =====
#!/usr/bin/env python3
"""
Import Time Budget Checker

Measures import time for critical modules and ensures they meet budget requirements.
Import time affects startup performance and user experience.

Budget: ≤ 300 ms per critical module (as specified in problem statement)

Exit codes:
- 0: All imports within budget
- 1: Some imports exceed budget
"""

from __future__ import annotations

import importlib
import sys
import time
from dataclasses import dataclass
from pathlib import Path


@dataclass
class ImportTiming:
    """Result of import timing measurement."""

    module: str
    time_ms: float
    success: bool
    error: str = ""


def measure_import_time(module_name: str) -> ImportTiming:
    """
    Measure the time to import a module.

    Note: This measures import time with the module potentially already
    in sys.modules cache. For accurate first-import timing, run in a
    fresh Python process.

    Parameters
    ----------
```

```
    module_name : str
        Fully qualified module name

    Returns
    -------
    ImportTiming
        Timing result with success status
    """
    try:
        start = time.perf_counter()
        importlib.import_module(module_name)
        end = time.perf_counter()

        elapsed_ms = (end - start) * 1000
        return ImportTiming(module_name, elapsed_ms, True)

    except Exception as e:
        return ImportTiming(module_name, 0.0, False, str(e))


def check_import_budget(budget_ms: float = 300.0) -> list[ImportTiming]:
    """
    Check import times for critical modules against budget.

    Parameters
    ----------
    budget_ms : float, default=300.0
        Maximum allowed import time in milliseconds

    Returns
    -------
    list[ImportTiming]
        Results for all tested modules
    """
    # Critical modules to test (from pyproject.toml and known heavy imports)
    critical_modules = [
        "saaaaaa",
        "saaaaaa.core",
        "saaaaaa.core.orchestrator",
        "saaaaaa.processing",
        "saaaaaa.processing.document_ingestion",
        "saaaaaa.analysis",
        "saaaaaa.concurrency",
        "saaaaaa.utils",
        "saaaaaa.compat",
    ]

    # Heavy optional dependencies to measure separately
    optional_modules = [
        "numpy",
        "pandas",
        "polars",
        "pyarrow",
        "torch",
        "tensorflow",
        "transformers",
        "spacy",
    ]

    results = []

    print("=== Import Budget Check ===")
    print(f"Budget: {budget_ms} ms per module\n")

    print("Critical Modules:")
    for module in critical_modules:
        result = measure_import_time(module)
        results.append(result)
```

```
        if result.success:
            status = "✓" if result.time_ms <= budget_ms else "✗"
            print(f"  {status} {module}: {result.time_ms:.1f} ms")
        else:
            print(f"  ✗ {module}: FAILED ({result.error})")

    print("\nOptional Dependencies (informational):")
    for module in optional_modules:
        result = measure_import_time(module)

        if result.success:
            print(f"    {module}: {result.time_ms:.1f} ms")
        else:
            print(f"    {module}: not installed or failed")

    return results


def main() -> int:
    """Main entry point."""
    results = check_import_budget()

    # Check for budget violations
    budget_ms = 300.0
    violations = [
        r for r in results
        if r.success and r.time_ms > budget_ms
    ]

    failures = [r for r in results if not r.success]

    print("\n=== Summary ===")
    print(f"Tested: {len(results)} modules")
    print(f"Budget violations: {len(violations)}")
    print(f"Import failures: {len(failures)}")

    if violations:
        print("\nModules exceeding budget:")
        for r in violations:
            print(f"  {r.module}: {r.time_ms:.1f} ms (over by {r.time_ms - budget_ms:.1f}
ms)")
        print("\nRecommendation: Apply lazy imports to reduce startup time")

    if failures:
        print("\nImport failures:")
        for r in failures:
            print(f"  {r.module}: {r.error}")

    if violations or failures:
        return 1

    print("\n✓ All imports within budget")
    return 0


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

===== FILE: scripts/audit_import_shadowing.py =====
#!/usr/bin/env python3
"""
Import Shadowing Detector

Detects local files that shadow standard library or third-party packages.
This is a critical security and correctness issue.

Examples of problems:
```

```
- json.py in the project shadows stdlib json
- typing.py shadows stdlib typing
- requests.py shadows third-party requests

Exit codes:
- 0: No shadowing detected
- 1: Shadowing detected (with details)
"""

from __future__ import annotations

import sys
from pathlib import Path


# Standard library modules that must not be shadowed
STDLIB_MODULES = {
    "abc", "ast", "asyncio", "base64", "collections", "concurrent",
    "contextlib", "copy", "csv", "dataclasses", "datetime", "decimal",
    "enum", "functools", "hashlib", "heapq", "http", "importlib",
    "inspect", "io", "itertools", "json", "logging", "math", "multiprocessing",
    "operator", "os", "pathlib", "pickle", "platform", "queue", "random",
    "re", "shutil", "signal", "socket", "sqlite3", "ssl", "string",
    "struct", "subprocess", "sys", "tempfile", "threading", "time",
    "traceback", "typing", "unittest", "urllib", "uuid", "warnings",
    "weakref", "xml", "zipfile", "zlib",
}

# Common third-party packages that must not be shadowed
THIRDPARTY_MODULES = {
    "numpy", "pandas", "scipy", "sklearn", "torch", "tensorflow",
    "requests", "httpx", "flask", "fastapi", "pydantic", "click",
    "pytest", "hypothesis", "black", "ruff", "mypy", "pyyaml",
    "polars", "pyarrow", "blake3", "networkx", "spacy", "transformers",
}


def find_python_files(root: Path) -> list[Path]:
    """Find all Python files in the project, excluding known safe directories."""
    exclude_patterns = {
        ".git", ".venv", "venv", "__pycache__", ".pytest_cache",
        ".mypy_cache", ".ruff_cache", "node_modules", "minipdm",
        "dist", "build", "*.egg-info",
    }

    python_files = []
    for py_file in root.rglob("*.py"):
        # Check if any excluded pattern is in the path
        if any(pattern in py_file.parts for pattern in exclude_patterns):
            continue
        python_files.append(py_file)

    return python_files


def check_shadowing(root: Path) -> list[tuple[Path, str, str]]:
    """
    Check for files that shadow stdlib or third-party modules.

    Returns
    -------
    list[tuple[Path, str, str]]
        List of (file_path, shadowed_module, category)
    """
    issues = []
    python_files = find_python_files(root)

    for py_file in python_files:
```

```
        # Get module name from filename
        stem = py_file.stem

        # Check against stdlib
        if stem in STDLIB_MODULES:
            issues.append((py_file, stem, "stdlib"))

        # Check against third-party
        elif stem in THIRDPARTY_MODULES:
            issues.append((py_file, stem, "third-party"))

    return issues


def main() -> int:
    """Main entry point."""
    root = Path(__file__).parent.parent

    print("=== Import Shadowing Detection ===")
    print(f"Scanning: {root}")
    print()

    issues = check_shadowing(root)

    if not issues:
        print("✓ No shadowing issues detected")
        return 0

    print(f"✗ Found {len(issues)} shadowing issue(s):\n")

    for file_path, module_name, category in sorted(issues):
        rel_path = file_path.relative_to(root)
        print(f"  {rel_path}")
        print(f"    Shadows {category} module: {module_name}")
        print(f"    Fix: Rename file to avoid import hijacking")
        print()

    print(f"Total issues: {len(issues)}")
    print("\nShadowing is a critical security and correctness issue.")
    print("Files must be renamed before they can be safely imported.")

    return 1


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

===== FILE: scripts/audit_paths.py =====
#!/usr/bin/env python3
"""
Comprehensive path audit script for SAAAAAA.

Scans the entire repository for path-related patterns and generates
a detailed audit report identifying risks and violations.
"""

from __future__ import annotations

import ast
import re
import sys
from collections import defaultdict
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

try:
    from saaaaaa.config.paths import PROJECT_ROOT as REPO_ROOT
```

```
except Exception:  # pragma: no cover - fallback when package not installed
    REPO_ROOT = Path(__file__).resolve().parents[1]


@dataclass
class PathFinding:
    """A single path-related finding."""
    file: Path
    line_number: int
    line_content: str
    category: str
    severity: str  # "critical", "high", "medium", "low"
    message: str
    fix_suggestion: str = ""


@dataclass
class PathAuditReport:
    """Complete path audit report."""
    findings: list[PathFinding] = field(default_factory=list)
    file_count: int = 0
    scanned_files: list[Path] = field(default_factory=list)
    stats: dict[str, Any] = field(default_factory=dict)

    def add_finding(self, finding: PathFinding) -> None:
        """Add a finding to the report."""
        self.findings.append(finding)

    def by_severity(self, severity: str) -> list[PathFinding]:
        """Get findings by severity level."""
        return [f for f in self.findings if f.severity == severity]

    def by_category(self, category: str) -> list[PathFinding]:
        """Get findings by category."""
        return [f for f in self.findings if f.category == category]

    def summary_stats(self) -> dict[str, int]:
        """Generate summary statistics."""
        return {
            "total_findings": len(self.findings),
            "critical": len(self.by_severity("critical")),
            "high": len(self.by_severity("high")),
            "medium": len(self.by_severity("medium")),
            "low": len(self.by_severity("low")),
            "files_scanned": self.file_count,
        }


class PathAuditor:
    """Main path auditor class."""

    def __init__(self, repo_root: Path):
        self.repo_root = repo_root
        self.report = PathAuditReport()

        # Patterns to detect various path issues
        self.patterns = {
            "sys_path_append": re.compile(r'sys\.path\.(append|insert)'),
            "absolute_unix":
re.compile(r'["\'](?:/home|/Users|/tmp|/var|/usr)/[^"\']*["\']'),
            "absolute_windows": re.compile(r'["\'][A-Z]:\\\\[^"\']*["\']'),
            "file_usage": re.compile(r'Path\s*\(\s*__file__\s*\)'),
            "file_parent": re.compile(r'__file__.*\.parent'),
            "open_builtin": re.compile(r'\bopen\s*\([^)]*\)'),
            "os_path_join": re.compile(r'os\.path\.join'),
            "os_path_exists": re.compile(r'os\.path\.exists'),
            "os_path_dirname": re.compile(r'os\.path\.dirname'),
            "os_path_abspath": re.compile(r'os\.path\.abspath'),
```

```
            "glob_usage": re.compile(r'\bglob\.(?:glob|iglob)'),
            "hardcoded_separator": re.compile(r'["\'][^"\']*[/\\]{2,}[^"\']*["\']'),
            "cwd_usage": re.compile(r'os\.getcwd\(\)|Path\.cwd\(\)'),
            "home_env": re.compile(r'os\.getenv\(["\']HOME["\']'),
            "temp_hardcode": re.compile(r'["\'](?:/tmp|C:\\\\temp)["\']'),
        }

    def should_skip(self, file_path: Path) -> bool:
        """Check if file should be skipped."""
        skip_patterns = [
            "/.venv/", "/venv/", "/env/",
            "/__pycache__/",
            "/minipdm/",
            "/.git/",
            "/node_modules/",
            ".pyc",
        ]

        file_str = str(file_path)
        return any(pattern in file_str for pattern in skip_patterns)

    def scan_file(self, file_path: Path) -> None:
        """Scan a single Python file for path issues."""
        if self.should_skip(file_path):
            return

        try:
            content = file_path.read_text(encoding='utf-8')
            lines = content.split('\n')

            rel_path = file_path.relative_to(self.repo_root)

            for line_num, line in enumerate(lines, 1):
                self._check_line(rel_path, line_num, line)

            # AST-based checks
            try:
                tree = ast.parse(content, str(file_path))
                self._check_ast(rel_path, tree, lines)
            except SyntaxError:
                pass  # Skip files with syntax errors

        except Exception as e:
            # Don't fail the whole scan on one bad file
            pass

    def _check_line(self, rel_path: Path, line_num: int, line: str) -> None:
        """Check a single line for path issues."""

        # sys.path manipulation
        if self.patterns["sys_path_append"].search(line):
            # Allow in scripts/ and tests/, but warn elsewhere
            if not (str(rel_path).startswith("scripts/") or
                    str(rel_path).startswith("tests/") or
                    str(rel_path).startswith("examples/")):
                self.report.add_finding(PathFinding(
                    file=rel_path,
                    line_number=line_num,
                    line_content=line.strip(),
                    category="sys_path_manipulation",
                    severity="critical",
                    message="sys.path manipulation detected outside
scripts/tests/examples",
                    fix_suggestion="Use proper package imports instead of sys.path"
                ))

        # Absolute paths (Unix-style)
        if self.patterns["absolute_unix"].search(line):
```

```
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="absolute_path",
                severity="high",
                message="Absolute Unix path detected",
                fix_suggestion="Use proj_root() or data_dir() from saaaaaa.utils.paths"
            ))

        # Absolute paths (Windows-style)
        if self.patterns["absolute_windows"].search(line):
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="absolute_path",
                severity="high",
                message="Absolute Windows path detected",
                fix_suggestion="Use proj_root() or data_dir() from saaaaaa.utils.paths"
            ))

        # __file__ usage for paths
        if self.patterns["file_usage"].search(line) or
self.patterns["file_parent"].search(line):
            # This is OK in scripts/ and specific places, but should be reviewed
            if not str(rel_path).startswith("scripts/"):
                self.report.add_finding(PathFinding(
                    file=rel_path,
                    line_number=line_num,
                    line_content=line.strip(),
                    category="file_usage",
                    severity="medium",
                    message="__file__ usage detected - may break in packaged
distributions",
                    fix_suggestion="Use resources() for packaged data or proj_root() for
workspace-relative paths"
                ))

        # os.path usage
        if (self.patterns["os_path_join"].search(line) or
            self.patterns["os_path_dirname"].search(line) or
            self.patterns["os_path_abspath"].search(line)):
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="os_path_usage",
                severity="medium",
                message="os.path usage detected",
                fix_suggestion="Use pathlib.Path instead of os.path"
            ))

        # hardcoded path separators
        if self.patterns["hardcoded_separator"].search(line):
            stripped = line.strip().lower()
            if "http://" in stripped or "https://" in stripped or ":// " in stripped:
                return
            if stripped.startswith("#"):
                return
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="hardcoded_separator",
                severity="medium",
                message="Potential hardcoded path separator detected",
                fix_suggestion="Use Path.joinpath() or / operator"
```

```
            ))

        # os.getcwd / current path usage
        if self.patterns["cwd_usage"].search(line):
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="cwd_usage",
                severity="medium",
                message="Current working directory usage - fragile in different execution
contexts",
                fix_suggestion="Use proj_root() or explicit paths from
saaaaaa.utils.paths"
            ))

        # HOME environment variable
        if self.patterns["home_env"].search(line):
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="home_env",
                severity="low",
                message="HOME environment variable usage",
                fix_suggestion="Use Path.home() from pathlib"
            ))

        # Hardcoded temp directories
        if self.patterns["temp_hardcode"].search(line):
            self.report.add_finding(PathFinding(
                file=rel_path,
                line_number=line_num,
                line_content=line.strip(),
                category="temp_hardcode",
                severity="high",
                message="Hardcoded temp directory path",
                fix_suggestion="Use tmp_dir() from saaaaaa.utils.paths or tempfile module"
            ))

    def _check_ast(self, rel_path: Path, tree: ast.AST, lines: list[str]) -> None:
        """Perform AST-based checks."""
        # Check for open() calls without proper context managers
        for node in ast.walk(tree):
            if isinstance(node, ast.Call):
                if isinstance(node.func, ast.Name) and node.func.id == "open":
                    # Check if it's inside a 'with' statement
                    # This is a simplified check - full check would need parent tracking
                    if hasattr(node, "lineno"):
                        line_content = lines[node.lineno - 1] if node.lineno <= len(lines)
 else ""
                        if "with" not in line_content:
                            # This is just informational - not all open() needs 'with'
                            self.report.add_finding(PathFinding(
                                file=rel_path,
                                line_number=node.lineno,
                                line_content=line_content.strip(),
                                category="open_without_context",
                                severity="low",
                                message="open() call - verify proper resource cleanup",
                                fix_suggestion="Consider using 'with open(...)' for
automatic cleanup"
                            ))

    def scan_repository(self) -> None:
        """Scan the entire repository."""
        print(f"Scanning repository: {self.repo_root}")
```

```
        # Find all Python files
        python_files = list(self.repo_root.rglob("*.py"))
        python_files = [f for f in python_files if not self.should_skip(f)]

        self.report.file_count = len(python_files)
        self.report.scanned_files = python_files

        print(f"Found {len(python_files)} Python files to scan")

        for i, py_file in enumerate(python_files, 1):
            if i % 50 == 0:
                print(f"  Scanned {i}/{len(python_files)} files...")
            self.scan_file(py_file)

        print(f"Scan complete. Found {len(self.report.findings)} issues.")

    def generate_markdown_report(self, output_path: Path) -> None:
        """Generate a detailed Markdown report."""
        stats = self.report.summary_stats()

        lines = [
            "# Path Audit Report",
            "",
            "**Generated by:** `scripts/audit_paths.py`",
            f"**Repository:** {self.repo_root}",
            "",
            "## Executive Summary",
            "",
            f"- **Files Scanned:** {stats['files_scanned']}",
            f"- **Total Findings:** {stats['total_findings']}",
            f"- **Critical:** {stats['critical']}",
            f"- **High:** {stats['high']}",
            f"- **Medium:** {stats['medium']}",
            f"- **Low:** {stats['low']}",
            "",
            "## Findings by Severity",
            "",
        ]

        # Group by severity
        for severity in ["critical", "high", "medium", "low"]:
            findings = self.report.by_severity(severity)
            if findings:
                lines.append(f"### {severity.upper()} ({len(findings)})")
                lines.append("")

                # Group by category
                by_category = defaultdict(list)
                for finding in findings:
                    by_category[finding.category].append(finding)

                for category, cat_findings in sorted(by_category.items()):
                    lines.append(f"#### {category} ({len(cat_findings)} occurrences)")
                    lines.append("")

                    # Show first 10 examples
                    for finding in cat_findings[:10]:
                        lines.append(f"- **{finding.file}:{finding.line_number}**")
                        lines.append(f"  - {finding.message}")
                        lines.append(f"  - Code: `{finding.line_content[:100]}`")
                        if finding.fix_suggestion:
                            lines.append(f"  - Fix: {finding.fix_suggestion}")
                        lines.append("")

                    if len(cat_findings) > 10:
                        lines.append(f"  ... and {len(cat_findings) - 10} more
occurrences")
                        lines.append("")
```

```
        # Category breakdown
        lines.extend([
            "## Findings by Category",
            "",
        ])

        by_category = defaultdict(list)
        for finding in self.report.findings:
            by_category[finding.category].append(finding)

        for category, findings in sorted(by_category.items(), key=lambda x: len(x[1]),
reverse=True):
            lines.append(f"### {category}: {len(findings)} occurrences")
            lines.append("")

        # Recommendations
        lines.extend([
            "",
            "## Recommendations",
            "",
            "1. **Eliminate sys.path manipulation**: Use proper package imports",
            "2. **Remove absolute paths**: Use `proj_root()`, `data_dir()`, etc. from
`saaaaaa.utils.paths`",
            "3. **Replace os.path with pathlib.Path**: More portable and Pythonic",
            "4. **Use resources() for packaged data**: Ensures compatibility with
wheels/sdist",
            "5. **Validate all path operations**: Use `validate_read_path()` and
`validate_write_path()`",
            "6. **Add path traversal protection**: Use `safe_join()` for user-provided
paths",
            "7. **Use temp_dir() instead of hardcoded /tmp**: Ensures controlled cleanup",
            "",
            "## Next Steps",
            "",
            "1. Create tests under `tests/paths/` to enforce these rules",
            "2. Fix critical and high severity issues first",
            "3. Add pre-commit hooks to prevent regressions",
            "4. Update CI to test on Windows, macOS, and Linux",
            "5. Document path handling guidelines in README",
            "",
        ])

        # Write report
        output_path.write_text("\n".join(lines), encoding='utf-8')
        print(f"\nReport written to: {output_path}")


def main() -> int:
    """Main entry point."""
    auditor = PathAuditor(REPO_ROOT)
    auditor.scan_repository()

    # Generate report
    output_path = REPO_ROOT / "PATHS_AUDIT.md"
    auditor.generate_markdown_report(output_path)

    # Print summary
    stats = auditor.report.summary_stats()
    print("\n" + "=" * 60)
    print("PATH AUDIT SUMMARY")
    print("=" * 60)
    print(f"Files scanned:     {stats['files_scanned']}")
    print(f"Total findings:    {stats['total_findings']}")
    print(f"  Critical:        {stats['critical']}")
    print(f"  High:            {stats['high']}")
    print(f"  Medium:          {stats['medium']}")
    print(f"  Low:             {stats['low']}")
```

```
    print("=" * 60)

    # Return non-zero if critical issues found
    return 1 if stats['critical'] > 0 else 0


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

===== FILE: scripts/audit_questionnaire_coverage.py =====
#!/usr/bin/env python3
"""
audit_questionnaire_coverage.py - Audit the structural and contract coverage of the
questionnaire.

This script performs a comprehensive audit of the questionnaire monolith and its
associated contracts and executors. It generates an audit manifest in JSON format
with detailed metrics about the coverage and any gaps found.

The audit checks for the following:
- Each micro-question in the monolith has a corresponding contract.
- Each contract has a corresponding micro-question.
- Each contract specifies method inputs.
- The classes and methods specified in the contract's method inputs are valid and
resolvable.
"""

import json
import os
import inspect
import sys
from pathlib import Path
from typing import Any, Dict, List, Set, Tuple

# Add src to python path
sys.path.append(str(Path(__file__).parent.parent / "src"))


# Assuming the script is in the 'scripts' directory, the project root is the parent
directory.
PROJECT_ROOT = Path(__file__).parent.parent.resolve()
MONOLITH_PATH = PROJECT_ROOT / "data" / "questionnaire_monolith.json"
CONTRACTS_DIR = PROJECT_ROOT / "config" / "executor_contracts"
CLASS_REGISTRY_PATH = PROJECT_ROOT / "src" / "saaaaaa" / "core" / "orchestrator" /
"class_registry.py"
OUTPUT_DIR = PROJECT_ROOT / "artifacts" / "audit"
AUDIT_MANIFEST_PATH = OUTPUT_DIR / "audit_manifest.json"

def get_micro_questions(monolith: Dict[str, Any]) -> List[Dict[str, Any]]:
    """
    Recursively finds and returns all micro-questions from the monolith.
    """
    micro_questions = []

    def find_in_obj(obj: Any):
        if isinstance(obj, dict):
            if "micro_questions" in obj and isinstance(obj["micro_questions"], list):
                micro_questions.extend(obj["micro_questions"])
            for key, value in obj.items():
                find_in_obj(value)
        elif isinstance(obj, list):
            for item in obj:
                find_in_obj(item)

    find_in_obj(monolith)
    return micro_questions

def get_contract_definitions() -> Dict[str, Dict[str, Any]]:
```

```
    """
    Loads all contract definitions from the contracts directory.
    The key of the returned dictionary is the base_slot.
    """
    contracts = {}
    if not CONTRACTS_DIR.is_dir():
        return contracts

    for contract_file in CONTRACTS_DIR.glob("*.json"):
        try:
            contract_data = json.loads(contract_file.read_text(encoding="utf-8"))
            base_slot = contract_data.get("base_slot")
            if base_slot:
                contracts[base_slot] = contract_data
        except (json.JSONDecodeError, KeyError) as e:
            print(f"Warning: Could not load or parse contract {contract_file}: {e}")
    return contracts

def get_registered_classes_and_methods() -> Dict[str, Set[str]]:
    """
    Dynamically loads the class registry and inspects the classes to get their methods.
    """
    # This is a simplified way to get the class paths. In a real scenario,
    # we would need to handle the imports and dependencies correctly.
    # For this script, we'll simulate by extracting from the file content.
    class_paths = {}
    with open(CLASS_REGISTRY_PATH, "r", encoding="utf-8") as f:
        registry_content = f.read()
        # A bit of a hack to extract the _CLASS_PATHS dictionary
        try:
            start = registry_content.find("_CLASS_PATHS: Mapping[str, str] = {")
            if start != -1:
                dict_str = registry_content[start:]
                dict_str = dict_str[dict_str.find("{") : dict_str.find("}") + 1]
                class_paths = eval(dict_str)
        except Exception as e:
            print(f"Warning: Could not parse class registry: {e}")
            return {}

    registered_methods = {}
    for class_name, import_path in class_paths.items():
        try:
            module_name, _, class_name_from_path = import_path.rpartition(".")
            module = __import__(module_name, fromlist=[class_name_from_path])
            cls = getattr(module, class_name_from_path)
            methods = {
                name
                for name, func in inspect.getmembers(cls, inspect.isfunction)
                if not name.startswith("_")
            }
            registered_methods[class_name] = methods
        except (ImportError, AttributeError) as e:
            print(f"Warning: Could not import or inspect class {class_name}: {e}")
    return registered_methods

def run_audit():
    """
    Runs the full audit and generates the manifest.
    """
    print("Starting questionnaire coverage audit...")

    # Load monolith
    if not MONOLITH_PATH.exists():
        print(f"Error: Monolith file not found at {MONOLITH_PATH}")
        return
    monolith = json.loads(MONOLITH_PATH.read_text(encoding="utf-8"))

    # Get data for audit
```

```
micro_questions = get_micro_questions(monolith)
contracts = get_contract_definitions()
registered_methods = get_registered_classes_and_methods()

# Audit metrics
total_micro_questions = len(micro_questions)
questions_with_contract = 0
questions_with_executor_mapping = 0
questions_with_valid_executor = 0
questions_with_valid_method_route = 0
orphan_contracts = set(contracts.keys())
uncontracted_questions_details = []
questions_with_contract = 0

for question in micro_questions:
    slot = question.get("base_slot")
    if slot and slot in contracts:
        questions_with_contract += 1
    elif slot:
        uncontracted_questions_details.append(question.get("question_id"))

questions_without_contract = len(uncontracted_questions_details)

questions_with_executor_mapping = 0
questions_with_valid_executor = 0
questions_with_valid_method_route = 0

orphan_contracts = set(contracts.keys())

processed_slots = set()
invalid_executor_questions = []
invalid_method_questions = []

for question in micro_questions:
    slot = question.get("base_slot")
    if not slot or slot in processed_slots:
        continue

    processed_slots.add(slot)
    orphan_contracts.discard(slot)

    if slot in contracts:
        contract = contracts[slot]
        method_inputs = contract.get("method_inputs", [])

        if method_inputs:
            questions_with_executor_mapping += 1

            all_executors_valid_for_slot = True
            all_methods_valid_for_slot = True

            for method_input in method_inputs:
                class_name = method_input.get("class")
                method_name = method_input.get("method")

                if class_name not in registered_methods:
                    all_executors_valid_for_slot = False
                    invalid_executor_questions.append({
                        "base_slot": slot,
                        "class": class_name
                    })
                elif method_name not in registered_methods.get(class_name, set()):
                    all_methods_valid_for_slot = False
                    invalid_method_questions.append({
                        "base_slot": slot,
                        "class": class_name,
                        "method": method_name
                    })
```

```
            if all_executors_valid_for_slot:
                questions_with_valid_executor += 1
            if all_methods_valid_for_slot and all_executors_valid_for_slot:
                questions_with_valid_method_route += 1

    # Prepare audit manifest
    audit_manifest = {
        "audit_timestamp": "2025-11-23T12:00:00Z",
        "metrics": {
            "total_micro_questions": total_micro_questions,
            "questions_with_contract": questions_with_contract,
            "questions_without_contract": questions_without_contract,
            "questions_with_executor_mapping": questions_with_executor_mapping,
            "questions_with_valid_executor": questions_with_valid_executor,
            "questions_with_valid_method_route": questions_with_valid_method_route,
            "orphan_contracts": len(orphan_contracts),
            "contract_coverage_percentage": (questions_with_contract /
total_micro_questions) * 100 if total_micro_questions > 0 else 0,
        },
        "gaps": {
            "questions_without_contract_details":
sorted(list(set(uncontracted_questions_details))),
            "orphan_contracts": sorted(list(orphan_contracts)),
            "questions_with_invalid_executor": invalid_executor_questions,
            "questions_with_invalid_method": invalid_method_questions,
        }
    }

    # Write manifest
    OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
    with open(AUDIT_MANIFEST_PATH, "w", encoding="utf-8") as f:
        json.dump(audit_manifest, f, indent=4, ensure_ascii=False)

    print(f"Audit complete. Manifest written to {AUDIT_MANIFEST_PATH}")
    print("\n--- Audit Metrics ---")
    print(json.dumps(audit_manifest["metrics"], indent=4))
    print("----------------------\n")

if __name__ == "__main__":
    run_audit()

===== FILE: scripts/audit_signal_coverage.py =====
#!/usr/bin/env python3
"""
audit_signal_coverage.py - Audit the coverage and effective consumption of signals.

This script performs an audit of policy areas and dimensions against available signal
definitions.
It generates a signal_audit_manifest.json with metrics such as:
- Total policy areas in the questionnaire.
- Policy areas with defined signals.
- Dimensions with defined signals.
- Orphan signals (signals without corresponding policy areas/dimensions).
"""

import json
import sys
from pathlib import Path
from typing import Any, Dict, List, Set, Tuple

# Add src to python path
sys.path.append(str(Path(__file__).parent.parent / "src"))

PROJECT_ROOT = Path(__file__).parent.parent.resolve()
MONOLITH_PATH = PROJECT_ROOT / "data" / "questionnaire_monolith.json"
POLICY_SIGNALS_DIR = PROJECT_ROOT / "config" / "policy_signals"
OUTPUT_DIR = PROJECT_ROOT / "artifacts" / "audit"
```

```
SIGNAL_AUDIT_MANIFEST_PATH = OUTPUT_DIR / "signal_audit_manifest.json"

def get_policy_areas_and_dimensions(monolith: Dict[str, Any]) -> Tuple[Set[str], Set[str],
 Dict[str, str]]:
    """
    Extracts all unique policy area IDs, dimension IDs, and a mapping from question_id to
dimension_id from the monolith.
    """
    policy_areas = set()
    dimensions = set()
    question_to_dimension_map = {}

    # Extract from canonical_notation
    canonical_notation = monolith.get("canonical_notation", {})
    for dim_key, dim_data in canonical_notation.get("dimensions", {}).items():
        dimensions.add(dim_data.get("code"))
    for pa_key, pa_data in canonical_notation.get("policy_areas", {}).items():
        policy_areas.add(pa_key)

    # Extract from micro_questions
    micro_questions = []
    def find_in_obj(obj: Any):
        if isinstance(obj, dict):
            if "micro_questions" in obj and isinstance(obj["micro_questions"], list):
                micro_questions.extend(obj["micro_questions"])
            for key, value in obj.items():
                find_in_obj(value)
        elif isinstance(obj, list):
            for item in obj:
                find_in_obj(item)
    find_in_obj(monolith)

    for q in micro_questions:
        pa_id = q.get("policy_area_id")
        dim_id = q.get("dimension_id")
        q_id = q.get("question_id")

        if pa_id:
            policy_areas.add(pa_id)
        if dim_id:
            dimensions.add(dim_id)
        if q_id and dim_id:
            question_to_dimension_map[q_id] = dim_id

    return policy_areas, dimensions, question_to_dimension_map

def get_signal_definitions() -> Dict[str, Dict[str, Any]]:
    """
    Loads all signal definitions from the policy_signals directory.
    The key of the returned dictionary is the policy area ID (e.g., "PA01").
    """
    signals = {}
    if not POLICY_SIGNALS_DIR.is_dir():
        return signals

    for signal_file in POLICY_SIGNALS_DIR.glob("*.json"):
        try:
            signal_data = json.loads(signal_file.read_text(encoding="utf-8"))
            # Assuming filename corresponds to policy area ID, e.g., PA01.json -> PA01
            pa_id = signal_file.stem
            signals[pa_id] = signal_data
        except (json.JSONDecodeError, KeyError) as e:
            print(f"Warning: Could not load or parse signal file {signal_file}: {e}",
file=sys.stderr)
    return signals

def run_audit():
    """
```

```
    Runs the signal coverage audit and generates the manifest.
    """
    print("Starting signal coverage audit...")

    # Load monolith
    if not MONOLITH_PATH.exists():
        print(f"Error: Monolith file not found at {MONOLITH_PATH}", file=sys.stderr)
        sys.exit(1)
    monolith = json.loads(MONOLITH_PATH.read_text(encoding="utf-8"))

    # Get data for audit
    all_policy_areas, all_dimensions, question_to_dimension_map =
get_policy_areas_and_dimensions(monolith)
    signal_definitions = get_signal_definitions()

    # Audit metrics
    total_pas_in_questionnaire = len(all_policy_areas)
    pas_with_signals = 0
    dimensions_with_signals_covered = set()
    orphan_signals = set(signal_definitions.keys())

    for pa_id in all_policy_areas:
        if pa_id in signal_definitions:
            pas_with_signals += 1
            orphan_signals.discard(pa_id)

            # Extract questions from signal metadata and map to dimensions
            signal_data = signal_definitions[pa_id]
            questions_in_signal = signal_data.get("metadata", {}).get("questions", [])
            for qid in questions_in_signal:
                if qid in question_to_dimension_map:
                    dimensions_with_signals_covered.add(question_to_dimension_map[qid])

    # Refine orphan signals: only if a PA has a signal file but that PA is not in the
monolith
    orphan_signals_not_in_monolith = [
        pa_id for pa_id in signal_definitions if pa_id not in all_policy_areas
    ]

    # Calculate coverage percentage for PAs
    pa_coverage_percentage = (pas_with_signals / total_pas_in_questionnaire) * 100 if
total_pas_in_questionnaire > 0 else 0

    # Prepare audit manifest
    signal_audit_manifest = {
        "audit_timestamp": "2025-11-23T12:00:00Z",
        "metrics": {
            "total_pas_in_questionnaire": total_pas_in_questionnaire,
            "pas_with_signals": pas_with_signals,
            "dimensions_with_signals": len(dimensions_with_signals_covered),
            "orphan_signals_not_in_monolith": len(orphan_signals_not_in_monolith),
            "pa_coverage_percentage": pa_coverage_percentage,
        },
        "gaps": {
            "pas_without_signals": sorted(list(all_policy_areas -
set(signal_definitions.keys()))),
            "orphan_signals_not_in_monolith": sorted(orphan_signals_not_in_monolith),
        }
    }

    # Write manifest
    OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
    with open(SIGNAL_AUDIT_MANIFEST_PATH, "w", encoding="utf-8") as f:
        json.dump(signal_audit_manifest, f, indent=4, ensure_ascii=False)

    print(f"Signal audit complete. Manifest written to {SIGNAL_AUDIT_MANIFEST_PATH}")
    print("\n--- Signal Audit Metrics ---")
    print(json.dumps(signal_audit_manifest["metrics"], indent=4))
```

```
    print("--------------------------\n")

if __name__ == "__main__":
    run_audit()

===== FILE: scripts/build_method_usage_intelligence.py =====
#!/usr/bin/env python3
"""
Method Usage Intelligence Scanner

Performs exhaustive codebase scan to build usage intelligence for every method:
- Count of usages across repo
- Processes/pipelines where it participates
- Execution topology (Solo, Sequential, Parallel, Interconnected)
- Parameterization locus (In-script, In YAML, In calibration_registry.py)

Output: Machine-readable metadata for auto-calibration decision system

Uses canonical method catalog: config/canonical_method_catalog.json (1,996 methods)
"""

import ast
import json
import re
import sys
from pathlib import Path
from collections import defaultdict
from typing import Dict, List, Set, Tuple, Optional
from dataclasses import dataclass, asdict

# Add src to path
repo_root = Path(__file__).parent.parent


@dataclass
class MethodUsage:
    """Usage intelligence for a single method"""
    class_name: str
    method_name: str
    fqn: str

    # Usage counts
    total_usages: int
    usage_locations: List[Dict[str, any]]  # [{file, line, context}, ...]

    # Pipeline participation
    pipelines: List[str]  # Names of pipelines/processes using this method

    # Execution topology
    execution_topology: str  # Solo, Sequential, Parallel, Interconnected

    # Parameterization
    param_in_script: bool  # Hardcoded in Python
    param_in_yaml: bool  # Configured via YAML (RED FLAG)
    param_in_registry: bool  # Configured via calibration_registry.py

    # Catalog status
    in_catalog: bool
    in_calibration_registry: bool

    # Criticality signals
    used_in_critical_path: bool
    method_priority: str  # From catalog
    method_complexity: str  # From catalog


class MethodUsageScanner:
    """Scans codebase for method usage patterns"""
```

```
    def __init__(self, repo_root: Path):
        self.repo_root = repo_root
        self.src_root = repo_root / "src"

        # Load canonical method catalog
        catalog_path = repo_root / "config" / "canonical_method_catalog.json"
        with open(catalog_path) as f:
            catalog_data = json.load(f)
        self.catalog_methods = catalog_data['methods']

        # Results
        self.usage_map: Dict[Tuple[str, str], MethodUsage] = {}
        self.calibration_registry_methods: Set[Tuple[str, str]] = set()

        # Patterns for detecting parameterization
        self.yaml_param_pattern = re.compile(r'\.yaml|\.yml', re.IGNORECASE)

    def scan(self):
        """Execute full usage scan"""
        print("="*80)
        print("METHOD USAGE INTELLIGENCE SCANNER")
        print("="*80)

        print("\n[1/5] Loading calibration registry methods...")
        self._load_calibration_registry()

        print(f"\n[2/5] Scanning Python files in {self.src_root}...")
        self._scan_python_files()

        print("\n[3/5] Scanning YAML configuration files...")
        self._scan_yaml_files()

        print("\n[4/5] Analyzing execution topology...")
        self._analyze_topology()

        print("\n[5/5] Building usage intelligence map...")
        self._build_usage_intelligence()

        print("\n✓ Scan complete!")

    def _load_calibration_registry(self):
        """Load methods from calibration_registry.py"""
        registry_path = self.repo_root / "src" / "saaaaaa" / "core" / "orchestrator" /
"calibration_registry.py"

        if not registry_path.exists():
            print(f"WARNING: calibration_registry.py not found at {registry_path}")
            return

        with open(registry_path, 'r', encoding='utf-8') as f:
            content = f.read()

        # Parse the CALIBRATIONS dict using AST
        tree = ast.parse(content, filename=str(registry_path))
        for node in ast.walk(tree):
            if isinstance(node, ast.Assign):
                for target in node.targets:
                    if isinstance(target, ast.Name) and target.id == 'CALIBRATIONS':
                        if isinstance(node.value, ast.Dict):
                            for key in node.value.keys:
                                # Expecting tuple of two strings: (class_name,
method_name)
                                if isinstance(key, ast.Tuple) and len(key.elts) == 2:
                                    elt0, elt1 = key.elts
                                    if isinstance(elt0, ast.Constant) and isinstance(elt1,
 ast.Constant):
                                        if isinstance(elt0.value, str) and
```

```
isinstance(elt1.value, str):

self.calibration_registry_methods.add((elt0.value, elt1.value))
        print(f"  Found {len(self.calibration_registry_methods)} methods in
calibration_registry.py")

    def _scan_python_files(self):
        """Scan Python files for method calls"""
        python_files = list(self.src_root.rglob("*.py"))
        print(f"  Found {len(python_files)} Python files")

        # Get catalog methods as a set for fast lookup
        catalog_method_set = {
            (m['class_name'], m['method_name'])
            for m in self.catalog_methods
            if m['class_name']
        }

        for py_file in python_files:
            try:
                with open(py_file, 'r', encoding='utf-8') as f:
                    content = f.read()

                tree = ast.parse(content)
                visitor = MethodCallVisitor(py_file, self.repo_root, catalog_method_set)
                visitor.visit(tree)

                # Collect results
                for (class_name, method_name), locations in visitor.method_calls.items():
                    key = (class_name, method_name)
                    if key not in self.usage_map:
                        self.usage_map[key] = {
                            'class_name': class_name,
                            'method_name': method_name,
                            'locations': []
                        }
                    self.usage_map[key]['locations'].extend(locations)

            except Exception as e:
                print(f"  ERROR parsing {py_file}: {e}")

        print(f"  Tracked {len(self.usage_map)} unique methods with actual usage")

    def _scan_yaml_files(self):
        """Scan YAML files for method references"""
        yaml_files = list(self.repo_root.rglob("*.yaml")) +
list(self.repo_root.rglob("*.yml"))
        print(f"  Found {len(yaml_files)} YAML files")

        for yaml_file in yaml_files:
            try:
                with open(yaml_file, 'r', encoding='utf-8') as f:
                    content = f.read()

                # Look for patterns that might reference methods
                # e.g., class: ClassName, method: method_name
                class_pattern = r'class:\s*([A-Za-z_][A-Za-z0-9_]*)'
                method_pattern = r'method:\s*([A-Za-z_][A-Za-z0-9_]*)'

                for line_num, line in enumerate(content.splitlines(), 1):
                    class_match = re.search(class_pattern, line)
                    method_match = re.search(method_pattern, line)

                    # Mark methods found in YAML
                    if class_match or method_match:
                        # This is a heuristic - need context to be sure
                        # For now, just flag files that have YAML config
                        pass
```

```
            except Exception as e:
                print(f"  ERROR reading {yaml_file}: {e}")

    def _analyze_topology(self):
        """Analyze execution topology of methods"""
        # For now, use simple heuristics
        # TODO: More sophisticated analysis of call graphs
        pass

    def _build_usage_intelligence(self):
        """Build final usage intelligence records"""
        all_catalog_methods = {(m.class_name, m.method_name): m for m in
self.catalog.all_methods()}

        # Build usage records for all catalogued methods
        for (class_name, method_name), catalog_method in all_catalog_methods.items():
            usage_data = self.usage_map.get((class_name, method_name), {})
            locations = usage_data.get('locations', [])

            usage = MethodUsage(
                class_name=class_name,
                method_name=method_name,
                fqn=catalog_method.fqn,
                total_usages=len(locations),
                usage_locations=locations,
                pipelines=[],  # TODO: extract from usage contexts
                execution_topology="Solo",  # TODO: analyze call patterns
                param_in_script=len(locations) > 0,  # If used, likely params in script
                param_in_yaml=False,  # TODO: detect from YAML scan
                param_in_registry=(class_name, method_name) in
self.calibration_registry_methods,
                in_catalog=True,
                in_calibration_registry=(class_name, method_name) in
self.calibration_registry_methods,
                used_in_critical_path=catalog_method.priority.value == "CRITICAL",
                method_priority=catalog_method.priority.value,
                method_complexity=catalog_method.complexity.value,
            )

            self.usage_map[(class_name, method_name)] = usage

        # Also track methods used but not in catalog (DEFECT)
        for (class_name, method_name), usage_data in list(self.usage_map.items()):
            if (class_name, method_name) not in all_catalog_methods:
                # Method used but not catalogued - this is a defect
                locations = usage_data.get('locations', [])
                usage = MethodUsage(
                    class_name=class_name,
                    method_name=method_name,
                    fqn=f"{class_name}.{method_name}",
                    total_usages=len(locations),
                    usage_locations=locations,
                    pipelines=[],
                    execution_topology="Unknown",
                    param_in_script=True,
                    param_in_yaml=False,
                    param_in_registry=False,
                    in_catalog=False,  # DEFECT
                    in_calibration_registry=False,
                    used_in_critical_path=False,
                    method_priority="UNKNOWN",
                    method_complexity="UNKNOWN",
                )
                self.usage_map[(class_name, method_name)] = usage

    def generate_report(self, output_path: Path):
        """Generate usage intelligence report"""
```

```
        report = {
            "metadata": {
                "generated_at": "2025-11-08",
                "total_methods_tracked": len(self.usage_map),
                "catalog_methods": self.catalog.total_methods,
                "calibration_registry_methods": len(self.calibration_registry_methods),
            },
            "methods": {}
        }

        # Convert usage records to dict
        for key, usage in self.usage_map.items():
            if isinstance(usage, MethodUsage):
                report["methods"][f"{usage.class_name}.{usage.method_name}"] =
asdict(usage)
            else:
                # Old dict format
                report["methods"][f"{key[0]}.{key[1]}"] = usage

        # Write report
        with open(output_path, 'w', encoding='utf-8') as f:
            json.dump(report, f, indent=2, ensure_ascii=False)

        print(f"\n✓ Usage intelligence report written to: {output_path}")

        # Print summary
        print("\n" + "="*80)
        print("USAGE INTELLIGENCE SUMMARY")
        print("="*80)

        in_catalog = sum(1 for u in self.usage_map.values() if isinstance(u, MethodUsage)
and u.in_catalog)
        not_in_catalog = sum(1 for u in self.usage_map.values() if isinstance(u,
MethodUsage) and not u.in_catalog)
        in_registry = sum(1 for u in self.usage_map.values() if isinstance(u, MethodUsage)
 and u.in_calibration_registry)

        print(f"Total methods tracked: {len(self.usage_map)}")
        print(f"  - In catalog: {in_catalog}")
        print(f"  - NOT in catalog (DEFECT): {not_in_catalog}")
        print(f"  - In calibration registry: {in_registry}")

        # Methods in catalog but never used
        unused = sum(1 for u in self.usage_map.values() if isinstance(u, MethodUsage) and
u.in_catalog and u.total_usages == 0)
        print(f"  - In catalog but NEVER used: {unused}")

        # Critical methods
        critical = sum(1 for u in self.usage_map.values() if isinstance(u, MethodUsage)
and u.used_in_critical_path)
        print(f"  - Critical methods: {critical}")


class MethodCallVisitor(ast.NodeVisitor):
    """AST visitor to extract method calls"""

    def __init__(self, file_path: Path, repo_root: Path, catalog_methods: Set[Tuple[str,
str]]):
        self.file_path = file_path
        self.repo_root = repo_root
        self.catalog_methods = catalog_methods
        self.method_calls: Dict[Tuple[str, str], List[dict]] = defaultdict(list)
        self.current_class = None
        self.imports = {}  # Track imports: {alias: module}
        self.class_instances = {}  # Track variable assignments to classes

    def visit_Import(self, node):
        """Track import statements"""
```

```
        for alias in node.names:
            name = alias.asname if alias.asname else alias.name
            self.imports[name] = alias.name
        self.generic_visit(node)

    def visit_ImportFrom(self, node):
        """Track from...import statements"""
        for alias in node.names:
            name = alias.asname if alias.asname else alias.name
            if node.module:
                self.imports[name] = f"{node.module}.{alias.name}"
            else:
                self.imports[name] = alias.name
        self.generic_visit(node)

    def visit_ClassDef(self, node):
        """Track current class context"""
        old_class = self.current_class
        self.current_class = node.name
        self.generic_visit(node)
        self.current_class = old_class

    def visit_Assign(self, node):
        """Track variable assignments that might be class instances"""
        try:
            if isinstance(node.value, ast.Call) and isinstance(node.value.func, ast.Name):
                class_name = node.value.func.id
                for target in node.targets:
                    if isinstance(target, ast.Name):
                        self.class_instances[target.id] = class_name
        except Exception:
            pass
        self.generic_visit(node)

    def visit_Call(self, node):
        """Extract method calls"""
        try:
            # Pattern 1: obj.method()
            if isinstance(node.func, ast.Attribute):
                method_name = node.func.attr
                class_name = None

                # Try to infer the class
                if isinstance(node.func.value, ast.Name):
                    # Direct call: obj.method()
                    obj_name = node.func.value.id

                    # Check if obj is a known class instance
                    if obj_name in self.class_instances:
                        class_name = self.class_instances[obj_name]
                    # Check if obj is a known import
                    elif obj_name in self.imports:
                        # Try to extract class name from import
                        import_path = self.imports[obj_name]
                        if '.' in import_path:
                            class_name = import_path.split('.')[-1]
                        else:
                            class_name = obj_name
                    # Check if it matches any catalog class
                    else:
                        for cat_class, cat_method in self.catalog_methods:
                            if method_name == cat_method:
                                # Potential match - use catalog class name
                                class_name = cat_class
                                break

                elif isinstance(node.func.value, ast.Call):
                    # Chained call: ClassName().method()
```

```
                if isinstance(node.func.value.func, ast.Name):
                    class_name = node.func.value.func.id

            # Record the call if we found a class
            if class_name and (class_name, method_name) in self.catalog_methods:
                location = {
                    'file': str(self.file_path.relative_to(self.repo_root)),
                    'line': node.lineno,
                    'context': 'method_call'
                }
                self.method_calls[(class_name, method_name)].append(location)

        except Exception:
            pass

        self.generic_visit(node)


def main():
    repo_root = Path(__file__).parent.parent
    scanner = MethodUsageScanner(repo_root)
    scanner.scan()

    output_path = repo_root / "config" / "method_usage_intelligence.json"
    scanner.generate_report(output_path)

    print("\n✓ Method usage intelligence scan complete!")


if __name__ == "__main__":
    main()

===== FILE: scripts/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

from saaaaaa.core.orchestrator import get_questionnaire_provider

QUESTIONNAIRE_PROVIDER = get_questionnaire_provider()

# Configure structured logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('forge.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)
```

```
class AbortError(Exception):
    """Fatal error requiring immediate abort."""
    def __init__(self, code: str, message: str, phase: str):
        self.code = code
        self.message = message
        self.phase = phase
        super().__init__(f"[{code}] {phase}: {message}")

@dataclass
class PhaseContext:
    """Context for a construction phase."""
    name: str
    preconditions: list[str]
    invariants: list[str]
    postconditions: list[str]

class MonolithForge:
    """
    Monolithic questionnaire builder following strict construction phases.
    """

    # Canonical constants
    CANONICAL_POLICY_AREAS = ['P1', 'P2', 'P3', 'P4', 'P5', 'P6', 'P7', 'P8', 'P9', 'P10']
    CANONICAL_DIMENSIONS = ['D1', 'D2', 'D3', 'D4', 'D5', 'D6']
    CANONICAL_SCORING_MODALITIES = ['TYPE_A', 'TYPE_B', 'TYPE_C', 'TYPE_D', 'TYPE_E',
'TYPE_F']

    # Quality thresholds for micro questions
    MICRO_QUALITY_LEVELS = {
        'EXCELENTE': 0.85,
        'BUENO': 0.70,
        'ACEPTABLE': 0.55,
        'INSUFICIENTE': 0.0
    }

    # Method sampling configuration
    METHODS_PER_BASE_SLOT = 2  # Number of methods to sample from catalog per base_slot

    def __init__(self):
        self.legacy_data = {}
        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():
            if count != 10:
                self.abort('A020', f'Base slot {slot} has {count} instances, expected 10',
 phase)

        # Verify we have exactly 30 base_slots
        if len(base_slot_counts) != 30:
            self.abort('A020', f'Expected 30 base_slots, found {len(base_slot_counts)}',
phase)

        logger.info(f"Mapped {len(base_slot_counts)} base_slots, each with 10 questions")
        logger.info(f"=== {phase} COMPLETE ===")

    # ============================================================================
    # PHASE 4: ExtractionAndNormalizationPhase
    # ============================================================================

    def extraction_and_normalization_phase(self):
        """
        Normalize field by field with strict validation.
        Invariants: text non-empty, scoring_modality valid
        """
        phase = "ExtractionAndNormalizationPhase"
        logger.info(f"=== {phase} START ===")

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

            # Normalize text (trim trailing whitespace, preserve internal)
            text = q.get('question_text', '').strip()
            if not text:
                self.abort('A030', f'Question {global_num} has empty text', phase)
            q['text'] = text

            # Validate scoring_modality
            scoring_modality = q.get('scoring_modality')
            if scoring_modality not in self.CANONICAL_SCORING_MODALITIES:
                self.abort('A030', f'Question {global_num} has invalid scoring_modality:
{scoring_modality}', phase)

            # Ensure no FIXME/TODO/TEMP markers
            for marker in ['FIXME', 'TODO', 'TEMP', 'LEGACY']:
                if marker in str(q):
                    self.abort('A030', f'Question {global_num} contains forbidden marker:
{marker}', phase)

        logger.info("Normalized 300 micro questions")
        logger.info(f"=== {phase} COMPLETE ===")

    # ============================================================================
    # PHASE 5: IndicatorsAndEvidencePhase
    # ============================================================================

    def indicators_and_evidence_phase(self):
        """
        Separate expected_elements, patterns, validation_checks.
        Invariants: Exact count preservation, no silent duplication
        """
        phase = "IndicatorsAndEvidencePhase"
```

```
        logger.info(f"=== {phase} START ===")

        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 []

            # Extract validation_checks
            validations = q.get('validation_checks', {})
            if isinstance(validations, dict):
                # Keep as structured dictionary
                q['validations'] = validations
            else:
```

```
            q['validations'] = validations if validations else {}

        logger.info("Extracted indicators and evidence for 300 questions")
        logger.info(f"=== {phase} COMPLETE ===")

    # ========================================================================
    # PHASE 6: MethodSetSynthesisPhase
    # ========================================================================

    def method_set_synthesis_phase(self):
        """
        Insert method_sets per base_slot from method catalog.
        Invariants: Each method has class, function, description, priority 1-3
        """
        phase = "MethodSetSynthesisPhase"
        logger.info(f"=== {phase} START ===")

        # Load real method catalog from canonical source
        from saaaaaa.core.orchestrator.factory import load_catalog

        # Load canonical method catalog (1,996 methods)
        catalog_path = self.repo_root / "config" / "canonical_method_catalog.json"
        try:
            catalog = load_catalog()
            logger.info(f"Loaded method catalog with {catalog.get('summary',
{}).get('total_methods', 0)} methods")
        except FileNotFoundError as e:
            self.abort('A050', f'Method catalog not found at {catalog_path}: {e}', phase)
        except Exception as e:
            self.abort(
                'A050',
                f'Failed to load method catalog: {e}\n'
                f'Check that the catalog file exists at {catalog_path} and is in the
expected format.',
                phase
            )

        # Extract methods from catalog and organize by base_slot
        base_slot_methods = {}

        # The catalog structure has files with methods
        files_data = catalog.get('files', {})

        # Create method sets per base_slot
        # Since the catalog doesn't directly map to base_slots, we create a
        # reasonable distribution of methods across base_slots
        for d_num in range(1, 7):  # D1-D6
            for q_num in range(1, 6):  # Q1-Q5
                base_slot = f"D{d_num}-Q{q_num}"

                # 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')

    for level_name, min_score in sorted(self.MICRO_QUALITY_LEVELS.items(), key=lambda
x: -x[1]):
        # Invariant: descending order
        if min_score >= prev_score:
            self.abort('A060', f'Rubric thresholds not descending: {level_name}',
phase)

        micro_levels.append({
            'level': level_name,
            'min_score': min_score,
            'color': self._get_level_color(level_name)
        })
        prev_score = min_score

    self.monolith['scoring_matrix'] = {
        'micro_levels': micro_levels,
        'modalities': scoring_matrix
    }

    logger.info(f"Transposed rubric with {len(micro_levels)} quality levels")
    logger.info(f"=== {phase} COMPLETE ===")

def _get_level_color(self, level_name: str) -> str:
    """Map quality level to color."""
    colors = {
        'EXCELENTE': 'green',
```

```
            'BUENO': 'blue',
            'ACEPTABLE': 'yellow',
            'INSUFICIENTE': 'red'
        }
        return colors.get(level_name, 'gray')

    # ========================================================================
    # PHASE 8: MesoMacroEmbeddingPhase
    # ========================================================================

    def meso_macro_embedding_phase(self):
        """
        Insert 4 MESO cluster questions and 1 MACRO question.
        Invariants: Clusters EXACT, hermeticity preserved
        """
        phase = "MesoMacroEmbeddingPhase"
        logger.info(f"=== {phase} START ===")

        # Verify cluster hermeticity BEFORE insertion
        questionnaire = self.legacy_data['questionnaire.json']
        legacy_clusters = questionnaire.get('metadata', {}).get('clusters', [])

        # Map legacy clusters to canonical
        cluster_mapping = {}
        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_record = self.canonical_clusters.get(cluster_id)

            if not canonical_record:
                self.abort('A070', f'Cluster {cluster_id} not found in canonical
registry', phase)

            canonical_areas = canonical_record.get('canonical', [])
            expected_legacy = canonical_record.get('legacy', [])

            if expected_legacy and set(legacy_areas) != set(expected_legacy):
                self.abort(
                    'A070',
                    f'{cluster_id} legacy hermeticity violation. '
                    f'Expected {expected_legacy}, got {legacy_areas}',
                    phase
                )

            cluster_mapping[cluster_id] = {
                'cluster_id': cluster_id,
                'policy_area_ids': canonical_areas,
                'legacy_policy_area_ids': legacy_areas,
                'label_es': cluster_def.get('i18n', {}).get('keys', {}).get('label_es',
''),
                'label_en': cluster_def.get('i18n', {}).get('keys', {}).get('label_en',
''),
                'rationale': cluster_def.get('rationale', '')
            }

        # Create 4 MESO questions (one per cluster)
        meso_questions = []

        for i, (cluster_id, cluster_info) in enumerate(sorted(cluster_mapping.items()),
301):
            meso_q = {
                'question_global': i,
                'question_id': f'MESO_{i-300}',
                'cluster_id': cluster_id,
                'type': 'MESO',
                'text': f"¿Cómo se integran las políticas en el cluster
{cluster_info['label_es']}?",
                'policy_areas': cluster_info['policy_area_ids'],
```

```
            'scoring_modality': 'MESO_INTEGRATION',
            'aggregation_method': 'weighted_average',
            'patterns': [
                {
                    'type': 'cross_reference',
                    'description': f'Verificar referencias cruzadas entre áreas
{cluster_info["policy_area_ids"]}'
                },
                {
                    'type': 'coherence',
                    'description': 'Evaluar coherencia narrativa entre políticas del
cluster'
                }
            ]
        }
        meso_questions.append(meso_q)

    # Create 1 MACRO question
    macro_question = {
        'question_global': 305,
        'question_id': 'MACRO_1',
        'type': 'MACRO',
        'text': '¿El Plan de Desarrollo presenta una visión integral y coherente que
articula todos los clusters y dimensiones?',
        'scoring_modality': 'MACRO_HOLISTIC',
        'aggregation_method': 'holistic_assessment',
        '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 integración entre todos los clusters',
                'priority': 1
            },
            {
                'type': 'long_term_vision',
                'description': 'Evaluar visión de largo plazo y transformación
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]}...")
        logger.info(f"=== {phase} COMPLETE ===")

    def _structure_patterns(self, pattern_refs: list, question_id: str) -> list:
        """Structure pattern_refs as typed objects with categories."""
```

```
        structured = []

        for idx, pattern in enumerate(pattern_refs):
            if not isinstance(pattern, str):
                continue

            # Categorize patterns based on content
            category = self._categorize_pattern(pattern)

            structured.append({
                'id': f"PAT-{question_id}-{idx:03d}",
                'pattern': pattern,
                'category': category,
                'match_type': 'REGEX' if any(c in pattern for c in r'\\d.*+?[]()') else
'LITERAL',
                'flags': 'i',
                'confidence_weight': 0.85
            })

        return structured

    def _categorize_pattern(self, pattern: str) -> str:
        """Categorize a pattern based on its content."""
        pattern_lower = pattern.lower()

        if any(src in pattern_lower for src in ['dane', 'medicina legal', 'fiscalía',
'policía', 'sivigila', 'sispro']):
            return 'FUENTE_OFICIAL'
        elif any(ind in pattern_lower for ind in ['tasa', 'porcentaje', '%', 'indicador',
'cifra']):
            return 'INDICADOR'
        elif any(year in pattern for year in ['20\\d{2}', 'año', 'periodo']):
            return 'TEMPORAL'
        elif any(ent in pattern_lower for ent in ['departamental', 'municipal',
'territorial']):
            return 'TERRITORIAL'
        elif any(unit in pattern_lower for unit in ['por 100', 'por 1.000', 'por cada']):
            return 'UNIDAD_MEDIDA'
        else:
            return 'GENERAL'

    def _create_scoring_definitions(self):
        """Create explicit scoring modality definitions."""
        return {
            'micro_levels': self.monolith['scoring_matrix']['micro_levels'],
            'modalities': self.monolith['scoring_matrix']['modalities'],
            'modality_definitions': {
                'TYPE_A': {
                    'description': 'Count 4 elements and scale to 0-3',
                    'aggregation': 'presence_threshold',
                    'threshold': 0.7,
                    'failure_code': 'F-A-MIN'
                },
                'TYPE_B': {
                    'description': 'Count up to 3 elements, each worth 1 point',
                    'aggregation': 'binary_sum',
                    'max_score': 3,
                    'failure_code': 'F-B-MIN'
                },
                'TYPE_C': {
                    'description': 'Count 2 elements and scale to 0-3',
                    'aggregation': 'presence_threshold',
                    'threshold': 0.5,
                    'failure_code': 'F-C-MIN'
                },
                'TYPE_D': {
                    'description': 'Count 3 elements, weighted',
                    'aggregation': 'weighted_sum',
```

```
                'weights': [0.4, 0.3, 0.3],
                'failure_code': 'F-D-MIN'
            },
            'TYPE_E': {
                'description': 'Boolean presence check',
                'aggregation': 'binary_presence',
                'failure_code': 'F-E-MIN'
            },
            'TYPE_F': {
                'description': 'Continuous scale',
                'aggregation': 'normalized_continuous',
                'normalization': 'minmax',
                'failure_code': 'F-F-MIN'
            }
        }
    }

def _calculate_ruleset_hash(self, micro_questions: list) -> str:
    """Calculate deterministic hash of all patterns for reproducibility."""
    all_patterns = []

    for q in micro_questions:
        for pattern in q.get('patterns', []):
            if isinstance(pattern, dict):
                all_patterns.append(pattern.get('pattern', ''))
            else:
                all_patterns.append(str(pattern))

    # Sort for determinism
    all_patterns.sort()

    # Concatenate and hash
    patterns_str = '|'.join(all_patterns)
    return hashlib.sha256(patterns_str.encode('utf-8')).hexdigest()

def _map_priority(self, priority_str: str) -> int:
    """Map string priority to numeric priority (1-3)."""
    priority_map = {
        'CRITICAL': 1,
        'HIGH': 1,
        'MEDIUM': 2,
        'LOW': 3,
    }
    return priority_map.get(priority_str, 2)

def _get_methods_for_base_slot(self, base_slot: str, files_data: dict) -> list[dict]:
    """Get methods for a specific base_slot from catalog data.

    Args:
        base_slot: Base slot identifier (e.g., 'D1-Q1')
        files_data: Method catalog files data

    Returns:
        List of method definitions for this base_slot
    """
    # Extract dimension and question from base_slot
    # This is a simplified mapping - production code should use proper configuration
    methods = []

    # Get a subset of methods from the catalog for this base_slot
    # In a real implementation, this mapping should come from configuration
    for file_name, file_data in files_data.items():
        file_methods = file_data.get('methods', [])

        # Take first METHODS_PER_BASE_SLOT methods as a representative sample
        for method_info in file_methods[:self.METHODS_PER_BASE_SLOT]:
            methods.append({
                'class': method_info.get('class', 'UnknownClass'),
```

```
                'function': method_info.get('method_name', 'unknown_method'),
                'module': file_name,
                'method_type': 'analysis',
                'priority': self._map_priority(method_info.get('priority', 'MEDIUM')),
                'description': method_info.get('description', f"Analysis method for
{base_slot}")
            })

        # Only need a few methods per base_slot
        if len(methods) >= self.METHODS_PER_BASE_SLOT:
            break

    # Ensure we have at least one method
    if not methods:
        logger.warning(
            f"No analysis methods found in catalog for base_slot '{base_slot}'. "
            f"Using synthetic DefaultAnalyzer fallback. This may indicate a
configuration issue."
        )
        methods.append({
            'class': 'DefaultAnalyzer',
            'function': 'analyze',
            'module': 'default',
            'method_type': 'analysis',
            'priority': 2,
            'description': f'Default analysis for {base_slot}'
        })

    return methods

# ============================================================================
# PHASE 10: ValidationReportPhase
# ============================================================================

def validation_report_phase(self):
    """
    Validate all invariants before emission.
    Invariants: All legacy counters == destination counters
    """
    phase = "ValidationReportPhase"
    logger.info(f"=== {phase} START ===")

    monolith = self.monolith['final']

    # Validate question counts
    micro_count = len(monolith['blocks']['micro_questions'])
    meso_count = len(monolith['blocks']['meso_questions'])
    macro_count = 1
    total_count = micro_count + meso_count + macro_count

    if micro_count != 300:
        self.abort('A090', f'Expected 300 micro questions, got {micro_count}', phase)

    if meso_count != 4:
        self.abort('A090', f'Expected 4 meso questions, got {meso_count}', phase)

    if total_count != 305:
        self.abort('A090', f'Expected 305 total questions, got {total_count}', phase)

    # Validate base_slot coverage
    base_slot_counts = defaultdict(int)
    for q in monolith['blocks']['micro_questions']:
        base_slot_counts[q['base_slot']] += 1

    for slot, count in base_slot_counts.items():
        if count != 10:
            self.abort('A090', f'Base slot {slot} has {count} questions, expected 10',
 phase)
```

```
        if len(base_slot_counts) != 30:
            self.abort('A090', f'Expected 30 base_slots, got {len(base_slot_counts)}',
phase)

        # Validate cluster hermeticity
        clusters_in_monolith = monolith['blocks']['niveles_abstraccion']['clusters']
        for cluster_def in clusters_in_monolith:
            cluster_id = cluster_def.get('cluster_id')
            canonical_record = self.canonical_clusters.get(cluster_id)

            if not canonical_record:
                self.abort('A090', f'Cluster {cluster_id} missing from canonical
registry', phase)

            canonical_expected = set(canonical_record.get('canonical', []))
            canonical_present = set(cluster_def.get('policy_area_ids', []))
            if canonical_expected and canonical_present != canonical_expected:
                self.abort(
                    'A090',
                    f'{cluster_id} canonical hermeticity violation in final',
                    phase
                )

            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/check_directive_compliance.py =====
#!/usr/bin/env python3
"""Directive Compliance Checker

This script validates that the repository complies with ALL directive requirements
from the problem statement:

1. Universal method coverage - no filters, no exceptions
2. Single canonical catalog - no conceptual splits
3. Machine-readable calibration requirements
4. Complete calibration tracking (centralized vs embedded)
5. Transitional cases explicitly managed
6. Stage-based implementation tracking

Exit codes:
  0 - Full compliance
  1 - Compliance violations found
"""

import json
import sys
from dataclasses import dataclass
from pathlib import Path
from typing import List, Dict, Any


@dataclass
class ComplianceViolation:
    """A directive compliance violation."""
    severity: str  # "critical", "high", "medium", "low"
    requirement: str  # Which directive requirement
    description: str
    remediation: str
```

```
class DirectiveComplianceChecker:
    """Checker for directive compliance."""

    def __init__(self, repo_root: Path):
        self.repo_root = repo_root
        self.violations: List[ComplianceViolation] = []

        # Load artifacts
        self.catalog = self._load_catalog()
        self.appendix = self._load_appendix()
        self.calibration_registry = self._load_calibration_registry()

    def _load_catalog(self) -> Dict[str, Any]:
        """Load canonical method catalog."""
        catalog_path = self.repo_root / "config" / "canonical_method_catalog.json"
        if not catalog_path.exists():
            raise FileNotFoundError(f"Canonical catalog not found: {catalog_path}")

        with open(catalog_path) as f:
            return json.load(f)

    def _load_appendix(self) -> Dict[str, Any]:
        """Load embedded calibration appendix."""
        appendix_path = self.repo_root / "config" / "embedded_calibration_appendix.json"
        if not appendix_path.exists():
            return {'metadata': {'total_embedded': 0}, 'embedded_calibrations': []}

        with open(appendix_path) as f:
            return json.load(f)

    def _load_calibration_registry(self) -> str:
        """Load calibration registry source code."""
        registry_path = self.repo_root / "src" / "saaaaaa" / "core" / "orchestrator" /
"calibration_registry.py"
        if not registry_path.exists():
            return ""

        with open(registry_path) as f:
            return f.read()

    def check_all(self) -> bool:
        """Run all compliance checks."""
        print("=" * 80)
        print("DIRECTIVE COMPLIANCE CHECKER")
        print("=" * 80)
        print()

        # Check each requirement
        self.check_requirement_1_universal_coverage()
        self.check_requirement_2_mechanical_decidability()
        self.check_requirement_3_calibration_implementation()
        self.check_requirement_4_transitional_cases()
        self.check_requirement_5_stage_enforcement()

        # Report results
        self._report_results()

        return len([v for v in self.violations if v.severity in ["critical", "high"]]) ==
0

    def check_requirement_1_universal_coverage(self):
        """Requirement 1: Universal method coverage, no filters."""
        print("Checking Requirement 1: Universal Coverage...")

        # Check directive compliance flags
        compliance = self.catalog['metadata']['directive_compliance']
```

```
        if not compliance.get('universal_coverage'):
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 1 - Universal Coverage",
                description="universal_coverage flag not set to true",
                remediation="Rebuild catalog with universal coverage enabled"
            ))

        if not compliance.get('no_filters_applied'):
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 1 - Universal Coverage",
                description="no_filters_applied flag not set to true",
                remediation="Rebuild catalog without any filters"
            ))

        # Check for suspicious exclusions
        total_methods = self.catalog['summary']['total_methods']
        if total_methods < 1000:
            self.violations.append(ComplianceViolation(
                severity="high",
                requirement="Requirement 1 - Universal Coverage",
                description=f"Suspiciously low method count: {total_methods}",
                remediation="Verify all Python files are being scanned"
            ))

        # Check for fabricated splits
        if 'subset_catalog' in self.catalog or 'filtered_methods' in self.catalog:
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 1 - Universal Coverage",
                description="Fabricated catalog split detected",
                remediation="Remove conceptual splits - maintain single canonical catalog"
            ))

        print(f"  ✓ Total methods tracked: {total_methods}")

    def check_requirement_2_mechanical_decidability(self):
        """Requirement 2: Calibration requirements mechanically decidable."""
        print("Checking Requirement 2: Mechanical Decidability...")

        methods = self.catalog['methods']
        compliance = self.catalog['metadata']['directive_compliance']

        # Check all methods have requires_calibration flag
        missing_flag = 0
        for method in methods:
            if 'requires_calibration' not in method:
                missing_flag += 1

        if missing_flag > 0:
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 2 - Mechanical Decidability",
                description=f"{missing_flag} methods missing requires_calibration flag",
                remediation="Rebuild catalog to add calibration flags to all methods"
            ))

        # Check for undocumented heuristics
        if not compliance.get('machine_readable_flags'):
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 2 - Mechanical Decidability",
                description="machine_readable_flags not set to true",
                remediation="Ensure all calibration decisions are machine-readable"
            ))
```

```
        # Check for improvised eligibility criteria
        methods_requiring = len([m for m in methods if m.get('requires_calibration')])
        print(f"  ✓ Methods requiring calibration: {methods_requiring}/{len(methods)}")

    def check_requirement_3_calibration_implementation(self):
        """Requirement 3: Calibration implementation status tracking."""
        print("Checking Requirement 3: Calibration Implementation Tracking...")

        by_status = self.catalog['summary']['by_calibration_status']

        # Check all status categories exist
        required_statuses = {'centralized', 'embedded', 'none', 'unknown'}
        for status in required_statuses:
            if status not in by_status:
                self.violations.append(ComplianceViolation(
                    severity="critical",
                    requirement="Requirement 3 - Calibration Tracking",
                    description=f"Missing calibration status category: {status}",
                    remediation="Rebuild catalog with all calibration status categories"
                ))

        # Check embedded calibrations are tracked
        embedded_count = by_status.get('embedded', 0)
        appendix_count = self.appendix['metadata']['total_embedded']

        if embedded_count != appendix_count:
            self.violations.append(ComplianceViolation(
                severity="high",
                requirement="Requirement 3 - Calibration Tracking",
                description=f"Embedded count mismatch: catalog={embedded_count},
appendix={appendix_count}",
                remediation="Rebuild embedded calibration appendix"
            ))

        # Check centralized calibrations reference registry
        centralized = self.catalog['calibration_tracking'].get('centralized', [])
        for method in centralized[:10]:  # Sample
            if 'calibration_registry.py' not in method.get('calibration_location', ''):
                self.violations.append(ComplianceViolation(
                    severity="medium",
                    requirement="Requirement 3 - Calibration Tracking",
                    description=f"Centralized method {method['canonical_name']} doesn't
reference registry",
                    remediation="Update calibration_location to reference
calibration_registry.py"
                ))
                break

        print(f"  ✓ Centralized: {by_status.get('centralized', 0)}")
        print(f"  ✓ Embedded: {embedded_count}")
        print(f"  ✓ Unknown: {by_status.get('unknown', 0)}")

    def check_requirement_4_transitional_cases(self):
        """Requirement 4: Transitional cases explicitly managed."""
        print("Checking Requirement 4: Transitional Cases...")

        embedded_count = self.catalog['summary']['by_calibration_status'].get('embedded',
0)

        if embedded_count > 0:
            # Check appendix exists
            appendix_path = self.repo_root / "config" /
"embedded_calibration_appendix.json"
            if not appendix_path.exists():
                self.violations.append(ComplianceViolation(
                    severity="critical",
                    requirement="Requirement 4 - Transitional Cases",
                    description=f"{embedded_count} embedded calibrations but no appendix
```

```
found",
                remediation="Run detect_embedded_calibrations.py to create appendix"
            ))

        # Check markdown documentation exists
        md_path = self.repo_root / "config" / "embedded_calibration_appendix.md"
        if not md_path.exists():
            self.violations.append(ComplianceViolation(
                severity="medium",
                requirement="Requirement 4 - Transitional Cases",
                description="Migration appendix markdown documentation missing",
                remediation="Generate embedded_calibration_appendix.md"
            ))

        # Check all embedded have migration metadata
        embedded = self.catalog['calibration_tracking'].get('embedded', [])
        for method in embedded:
            if 'embedded_calibration' not in method:
                self.violations.append(ComplianceViolation(
                    severity="high",
                    requirement="Requirement 4 - Transitional Cases",
                    description=f"Embedded method {method['canonical_name']} missing
migration metadata",
                    remediation="Run detect_embedded_calibrations.py and update
catalog"
                ))
                break

        print(f"  ✓ Transitional cases tracked: {embedded_count}")

        # Check for critical priority items
        if embedded_count > 0:
            by_priority = self.appendix['metadata'].get('by_priority', {})
            critical = by_priority.get('critical', 0)
            high = by_priority.get('high', 0)

            if critical > 0:
                self.violations.append(ComplianceViolation(
                    severity="high",
                    requirement="Requirement 4 - Transitional Cases",
                    description=f"{critical} CRITICAL priority embedded calibrations need
immediate migration",
                    remediation="Migrate critical priority methods to
calibration_registry.py"
                ))

            if high > 5:
                self.violations.append(ComplianceViolation(
                    severity="medium",
                    requirement="Requirement 4 - Transitional Cases",
                    description=f"{high} HIGH priority embedded calibrations need
migration",
                    remediation="Plan migration for high priority methods"
                ))

    def check_requirement_5_stage_enforcement(self):
        """Requirement 5: Stage-based implementation enforcement."""
        print("Checking Requirement 5: Stage Enforcement...")

        # Check stage documentation exists
        stages_doc = self.repo_root / "IMPLEMENTATION_STAGES.md"
        if not stages_doc.exists():
            self.violations.append(ComplianceViolation(
                severity="high",
                requirement="Requirement 5 - Stage Enforcement",
                description="Implementation stages documentation missing",
                remediation="Create IMPLEMENTATION_STAGES.md with stage tracking"
            ))
```

```
        # Check canonical catalog is single source of truth
        if not
self.catalog['metadata']['directive_compliance'].get('single_canonical_source'):
            self.violations.append(ComplianceViolation(
                severity="critical",
                requirement="Requirement 5 - Stage Enforcement",
                description="Single canonical source flag not set",
                remediation="Ensure catalog is sole source of truth for methods"
            ))

        # Check for parallel math or hidden defaults
        unknown_count = self.catalog['summary']['by_calibration_status'].get('unknown', 0)
        total_requiring = len([m for m in self.catalog['methods'] if
m.get('requires_calibration')])

        if unknown_count > total_requiring * 0.3:  # More than 30% unknown
            self.violations.append(ComplianceViolation(
                severity="high",
                requirement="Requirement 5 - Stage Enforcement",
                description=f"High unknown calibration status:
{unknown_count}/{total_requiring} ({100*unknown_count/total_requiring:.1f}%)",
                remediation="Investigate unknown status methods to ensure no hidden
defaults"
            ))

        print(f"  ✓ Stage tracking: {stages_doc.exists()}")
        print(f"  ✓ Unknown status: {unknown_count}
({100*unknown_count/len(self.catalog['methods']):.1f}%)")

    def _report_results(self):
        """Report compliance results."""
        print()
        print("=" * 80)
        print("COMPLIANCE REPORT")
        print("=" * 80)
        print()

        if not self.violations:
            print("✓ FULL COMPLIANCE")
            print()
            print("All directive requirements met:")
            print("  ✓ Universal coverage - no filters or exceptions")
            print("  ✓ Mechanical decidability - all flags present")
            print("  ✓ Complete tracking - all calibrations visible")
            print("  ✓ Transitional cases - explicitly managed")
            print("  ✓ Stage enforcement - documented and tracked")
            return

        # Group violations by severity
        by_severity = {
            'critical': [],
            'high': [],
            'medium': [],
            'low': []
        }

        for violation in self.violations:
            by_severity[violation.severity].append(violation)

        # Report critical and high first
        critical_count = len(by_severity['critical'])
        high_count = len(by_severity['high'])

        if critical_count > 0 or high_count > 0:
            print(f"✗ COMPLIANCE VIOLATIONS: {critical_count} critical, {high_count}
high")
        else:
```

```
            print(f"⚠   MINOR ISSUES: {len(self.violations)} warnings")

        print()

        # Detail each severity level
        for severity in ['critical', 'high', 'medium', 'low']:
            violations = by_severity[severity]
            if not violations:
                continue

            icon = " ✗ " if severity in ['critical', 'high'] else "⚠ "
            print(f"{icon} {severity.upper()} ({len(violations)}):")
            print()

            for i, violation in enumerate(violations, 1):
                print(f"  {i}. {violation.requirement}")
                print(f"     Issue: {violation.description}")
                print(f"     Fix: {violation.remediation}")
                print()


def main():
    """Main entry point."""
    repo_root = Path(__file__).parent.parent

    try:
        checker = DirectiveComplianceChecker(repo_root)
        compliant = checker.check_all()

        return 0 if compliant else 1

    except FileNotFoundError as e:
        print(f"Error: {e}")
        print("\nRun these commands first:")
        print("  python3 scripts/build_canonical_method_catalog.py")
        print("  python3 scripts/detect_embedded_calibrations.py")
        return 1


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

===== FILE: scripts/check_version_pins.py =====
#!/usr/bin/env python3
"""
Check if dependencies have appropriate version constraints.

This script verifies that requirement files use exact pins (==) or properly
constrained ranges (>=X,<Y) for packages that need flexible dependency resolution.
Used in CI to enforce reproducible yet resolvable builds.
"""

import re
import sys
from pathlib import Path
from typing import List, Tuple


# Packages allowed to use constrained ranges due to complex dependency chains
ALLOWED_CONSTRAINED_RANGES = {
    'fastapi', 'huggingface-hub', 'numpy', 'pandas', 'pydantic',
    'safetensors', 'scikit-learn', 'scipy', 'sentence-transformers',
    'tokenizers', 'transformers'
}


def check_file_for_version_constraints(filepath: Path) -> Tuple[bool, List[str]]:
    """
```

```
    Check a requirements file for appropriate version constraints.

    Returns:
        Tuple of (has_violations, list_of_violations)
    """
    if not filepath.exists():
        return False, []

    violations = []
    # Match version specifiers more precisely: package_name OPERATOR version
    # Package names can contain letters, numbers, underscores, hyphens, and dots
    # NOTE: This regex parses Python package version specifiers (PEP 440), NOT HTML tags
    # CodeQL alert py/bad-tag-filter is a FALSE POSITIVE - we are not parsing HTML
    version_specifier_pattern = re.compile(r'^([a-zA-Z0-9_.-]+)\s*(>=|~=|<=|<|>|==|\*)')
# noqa: DUO138

    with open(filepath, 'r') as f:
        for line_num, line in enumerate(f, 1):
            original_line = line
            line = line.strip()

            # Skip empty lines, comments, and -r includes
            if not line or line.startswith('#') or line.startswith('-r '):
                continue

            # Check for version constraints
            match = version_specifier_pattern.match(line)
            if match:
                pkg_name = match.group(1).lower()
                operator = match.group(2)

                # Exact pins are always OK
                if operator == '==':
                    continue

                # Check for constrained ranges (>=X,<Y)
                if operator == '>=' and '<' in line:
                    # This is a constrained range
                    if pkg_name not in ALLOWED_CONSTRAINED_RANGES:
                        violations.append(
                            f"Line {line_num}: {original_line.strip()}\n"
                            f"    Package '{pkg_name}' not allowed to use constrained
ranges.\n"
                            f"    Use exact pin (==) or add to ALLOWED_CONSTRAINED_RANGES
if needed."
                        )
                    # Otherwise, it's allowed
                else:
                    # Unconstrained or unusual operator
                    violations.append(
                        f"Line {line_num}: {original_line.strip()}\n"
                        f"    Use exact pin (==) or constrained range (>=X,<Y)"
                    )

    return len(violations) > 0, violations


def main():
    """Main entry point."""
    if len(sys.argv) < 2:
        print("Usage: check_version_pins.py <requirement-file> [<requirement-file> ...]")
        return 1

    print("="*80)
    print("VERSION CONSTRAINT VALIDATOR")
    print("="*80)
    print("Checking for appropriate version constraints")
    print("Allowed: exact pins (==) or constrained ranges (>=X,<Y) for approved packages")
```

```
    print()

    total_violations = 0
    files_with_violations = []

    for filepath_str in sys.argv[1:]:
        filepath = Path(filepath_str)
        print(f"Checking {filepath}...")

        has_violations, violations = check_file_for_version_constraints(filepath)

        if has_violations:
            total_violations += len(violations)
            files_with_violations.append(filepath)
            print(f"  ✗ Found {len(violations)} violation(s):")
            for violation in violations:
                print(f"    {violation}")
        else:
            print(f"  ✓ All version constraints are appropriate")
        print()

    print("="*80)
    print("SUMMARY")
    print("="*80)
    print(f"Files checked: {len(sys.argv) - 1}")
    print(f"Files with violations: {len(files_with_violations)}")
    print(f"Total violations: {total_violations}")

    if total_violations > 0:
        print("\n✗ FAILED: Inappropriate version constraints detected!")
        print("\nFor reproducible builds, use exact pins (==) or constrained ranges
(>=X,<Y)")
        print("Example: numpy==2.2.1 or transformers>=4.40.0,<5.0.0")
        return 1
    else:
        print("\n✓ SUCCESS: All version constraints are appropriate")
        return 0


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

===== FILE: scripts/compare_freeze_lock.py =====
#!/usr/bin/env python3
"""
Compare pip freeze output with expected constraints/lock file.

This script is used in CI to ensure that installed packages match expected versions.
"""

import sys
from pathlib import Path
from typing import Dict, List, Set, Tuple


def parse_requirements_file(filepath: Path) -> Dict[str, str]:
    """Parse a requirements file and return package->version mapping."""
    packages = {}

    if not filepath.exists():
        return packages

    with open(filepath, 'r') as f:
        for line in f:
            line = line.strip()

            # Skip empty lines and comments
            if not line or line.startswith('#'):
```

```
            continue

        # Skip -r includes
        if line.startswith('-r '):
            continue

        # Parse package==version
        if '==' in line:
            pkg, version = line.split('==', 1)
            packages[pkg.lower().strip()] = version.strip()
        elif '>=' in line or '~=' in line or '<=' in line:
            # For now, skip range specifications
            continue

    return packages


def compare_packages(freeze: Dict[str, str], lock: Dict[str, str]) -> Tuple[Set[str],
Set[str], Dict[str, Tuple[str, str]]]:
    """
    Compare freeze and lock dictionaries.

    Returns:
        - missing_in_freeze: packages in lock but not in freeze
        - extra_in_freeze: packages in freeze but not in lock
        - version_mismatches: packages with different versions
    """
    freeze_keys = set(freeze.keys())
    lock_keys = set(lock.keys())

    missing_in_freeze = lock_keys - freeze_keys
    extra_in_freeze = freeze_keys - lock_keys

    version_mismatches = {}
    for pkg in freeze_keys & lock_keys:
        if freeze[pkg] != lock[pkg]:
            version_mismatches[pkg] = (freeze[pkg], lock[pkg])

    return missing_in_freeze, extra_in_freeze, version_mismatches


def main():
    """Main entry point."""
    if len(sys.argv) != 3:
        print("Usage: compare_freeze_lock.py <freeze-file> <lock-file>")
        print("  freeze-file: Output from 'pip freeze'")
        print("  lock-file: Expected constraints file")
        return 1

    freeze_file = Path(sys.argv[1])
    lock_file = Path(sys.argv[2])

    if not freeze_file.exists():
        print(f"Error: Freeze file not found: {freeze_file}")
        return 1

    if not lock_file.exists():
        print(f"Error: Lock file not found: {lock_file}")
        return 1

    print("="*80)
    print("FREEZE vs LOCK COMPARISON")
    print("="*80)
    print(f"Freeze file: {freeze_file}")
    print(f"Lock file: {lock_file}")
    print()

    freeze = parse_requirements_file(freeze_file)
```

```
    lock = parse_requirements_file(lock_file)

    print(f"Packages in freeze: {len(freeze)}")
    print(f"Packages in lock: {len(lock)}")
    print()

    missing, extra, mismatches = compare_packages(freeze, lock)

    has_errors = False

    # Report missing packages
    if missing:
        has_errors = True
        print("❌ MISSING IN FREEZE (in lock but not installed):")
        for pkg in sorted(missing):
            print(f"  - {pkg}=={lock[pkg]}")
        print()

    # Report extra packages (informational only)
    if extra:
        print("⚠️  EXTRA IN FREEZE (installed but not in lock):")
        for pkg in sorted(extra):
            print(f"  - {pkg}=={freeze[pkg]}")
        print("  (This may be OK if they are transitive dependencies)")
        print()

    # Report version mismatches
    if mismatches:
        has_errors = True
        print("❌ VERSION MISMATCHES:")
        for pkg, (freeze_ver, lock_ver) in sorted(mismatches.items()):
            print(f"  - {pkg}:")
            print(f"      Installed: {freeze_ver}")
            print(f"      Expected:  {lock_ver}")
        print()

    # Summary
    print("="*80)
    if not has_errors:
        print("✅ SUCCESS: Freeze matches lock file")
        return 0
    else:
        print("❌ FAILURE: Discrepancies detected")
        print()
        print("To fix:")
        print("  1. Install exact versions: pip install -c constraints-new.txt -r
requirements-core.txt")
        print("  2. Or regenerate lock: pip freeze > constraints-new.txt")
        return 1


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

===== FILE: scripts/comprehensive_pipeline_audit.py =====
#!/usr/bin/env python3
"""
Comprehensive Pipeline Technical Audit
======================================

Executes exhaustive audit of the complete pipeline (ingest → normalize → chunk →
signals → aggregate → score → report) detecting gaps, incompatibilities, technical
debt and operational risks with reproducible evidence.

Operating Mode: Deterministic, no silent heuristics
Exit Code: Non-zero if CRITICAL findings exist
"""
```

```
import ast
import importlib
import inspect
import json
import os
import re
import sys
from collections import defaultdict
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from pathlib import Path
from typing import Any, Dict, List, Optional, Set, Tuple

# Add src to path for imports


class Severity(Enum):
    """Finding severity levels."""
    CRITICAL = "CRITICAL"
    HIGH = "HIGH"
    MEDIUM = "MEDIUM"
    LOW = "LOW"
    INFO = "INFO"


@dataclass
class Finding:
    """Represents a single audit finding."""
    id: str
    severity: Severity
    category: str
    title: str
    description: str
    evidence: List[str] = field(default_factory=list)
    file_location: Optional[str] = None
    line_number: Optional[int] = None
    remediation: Optional[str] = None
    test_name: Optional[str] = None
    test_result: Optional[str] = None

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary."""
        return {
            "id": self.id,
            "severity": self.severity.value,
            "category": self.category,
            "title": self.title,
            "description": self.description,
            "evidence": self.evidence,
            "file_location": self.file_location,
            "line_number": self.line_number,
            "remediation": self.remediation,
            "test_name": self.test_name,
            "test_result": self.test_result,
        }


@dataclass
class AuditMetrics:
    """Audit execution metrics."""
    signal_hit_rate: float = 0.0
    signal_staleness_s: float = 0.0
    provenance_completeness: float = 0.0
    arg_router_routes_count: int = 0
    arg_router_silent_drops: int = 0
    determinism_phase_hashes_match: bool = False
```

```
    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary."""
        return {
            "signal_hit_rate": self.signal_hit_rate,
            "signal_staleness_s": self.signal_staleness_s,
            "provenance_completeness": self.provenance_completeness,
            "arg_router_routes_count": self.arg_router_routes_count,
            "arg_router_silent_drops": self.arg_router_silent_drops,
            "determinism_phase_hashes_match": self.determinism_phase_hashes_match,
        }


class ComprehensivePipelineAuditor:
    """Main auditor class."""

    def __init__(self, repo_root: Path):
        """Initialize auditor."""
        self.repo_root = repo_root
        self.src_root = repo_root / "src" / "saaaaaa"
        self.findings: List[Finding] = []
        self.metrics = AuditMetrics()
        self.finding_counter = 0

    def generate_finding_id(self, category: str) -> str:
        """Generate unique finding ID."""
        self.finding_counter += 1
        return f"{category.upper()}-{self.finding_counter:03d}"

    def add_finding(self, finding: Finding) -> None:
        """Add a finding to the list."""
        self.findings.append(finding)

    # ========================================================================
    # Phase 1: Contract Compatibility Analysis
    # ========================================================================

    def audit_contract_compatibility(self) -> None:
        """Validate Deliverable_i → Expectation_{i+1} contracts."""
        print("📋 Phase 1: Contract Compatibility Analysis")

        # Check for Pydantic schemas
        contracts_dir = self.repo_root / "contracts"
        schemas_dir = self.repo_root / "config" / "schemas"

        if not contracts_dir.exists() and not schemas_dir.exists():
            self.add_finding(Finding(
                id=self.generate_finding_id("CONTRACT"),
                severity=Severity.CRITICAL,
                category="Contract Compatibility",
                title="No contracts directory found",
                description="Neither contracts/ nor config/schemas/ directory exists",
                evidence=[
                    f"Expected: {contracts_dir}",
                    f"Expected: {schemas_dir}",
                ],
                remediation="Create contracts directory with Pydantic schemas for each
pipeline stage",
            ))
            return

        # Look for contract definitions
        contract_files = []
        if contracts_dir.exists():
            contract_files.extend(list(contracts_dir.glob("**/*.py")))
        if schemas_dir.exists():
            contract_files.extend(list(schemas_dir.glob("**/*.json")))

        print(f"  Found {len(contract_files)} contract files")
```

```
        # Check for specific pipeline stage contracts
        required_contracts = [
            "preprocessed_document",  # Ingest output
            "canonical_policy_package",  # CPP format
            "chunk_graph",  # Chunking output
            "signal_pack",  # Signal format
            "scored_result",  # Scoring output
        ]

        found_contracts = set()
        for contract_file in contract_files:
            content = contract_file.read_text(errors='ignore')
            for contract_name in required_contracts:
                if contract_name.replace("_", "") in content.lower().replace("_", ""):
                    found_contracts.add(contract_name)

        missing_contracts = set(required_contracts) - found_contracts
        if missing_contracts:
            self.add_finding(Finding(
                id=self.generate_finding_id("CONTRACT"),
                severity=Severity.HIGH,
                category="Contract Compatibility",
                title="Missing pipeline stage contracts",
                description=f"Missing contracts for: {',
'.join(sorted(missing_contracts))}",
                evidence=[f"Required: {c}" for c in sorted(missing_contracts)],
                remediation="Define Pydantic schemas for all pipeline stage interfaces",
            ))

        # Check for Pydantic usage in contracts
        pydantic_found = False
        for py_file in contract_files:
            if py_file.suffix == ".py":
                content = py_file.read_text(errors='ignore')
                if "pydantic" in content.lower() or "BaseModel" in content:
                    pydantic_found = True
                    break

        if not pydantic_found and any(f.suffix == ".py" for f in contract_files):
            self.add_finding(Finding(
                id=self.generate_finding_id("CONTRACT"),
                severity=Severity.MEDIUM,
                category="Contract Compatibility",
                title="Pydantic not used for contract validation",
                description="No Pydantic BaseModel found in contract definitions",
                evidence=["Searched in contracts/ and config/schemas/"],
                remediation="Use Pydantic BaseModel for all contract schemas to ensure
type safety",
            ))

    # ========================================================================
    # Phase 2: Parametrization Audit
    # ========================================================================

    def audit_parametrization(self) -> None:
        """Verify Config classes with from_env() and from_cli()."""
        print("🔧 Phase 2: Parametrization Audit")

        # Find all Python files in src
        py_files = list(self.src_root.glob("**/*.py"))

        config_classes_found = 0
        config_classes_with_from_env = 0
        config_classes_with_from_cli = 0

        for py_file in py_files:
            try:
```

```
            content = py_file.read_text()
            tree = ast.parse(content)

            for node in ast.walk(tree):
                if isinstance(node, ast.ClassDef):
                    # Check if it's a Config class
                    if "Config" in node.name or "config" in node.name.lower():
                        config_classes_found += 1

                        # Check for from_env and from_cli methods
                        methods = [m.name for m in node.body if isinstance(m,
ast.FunctionDef)]
                        has_from_env = "from_env" in methods
                        has_from_cli = "from_cli" in methods

                        if has_from_env:
                            config_classes_with_from_env += 1
                        if has_from_cli:
                            config_classes_with_from_cli += 1

                        if not (has_from_env and has_from_cli):
                            self.add_finding(Finding(
                                id=self.generate_finding_id("PARAM"),
                                severity=Severity.MEDIUM,
                                category="Parametrization",
                                title=f"Config class missing standard methods:
{node.name}",
                                description=f"Config class lacks from_env={'✓' if
has_from_env else '✗'} or from_cli={'✓' if has_from_cli else '✗'}",

file_location=str(py_file.relative_to(self.repo_root)),
                                line_number=node.lineno,
                                evidence=[
                                    f"Class: {node.name}",
                                    f"Has from_env: {has_from_env}",
                                    f"Has from_cli: {has_from_cli}",
                                ],
                                remediation=f"Add {'from_env' if not has_from_env else
 ''}{' and ' if not (has_from_env or has_from_cli) else ''}{'from_cli' if not has_from_cli
 else ''} methods to {node.name}",
                            ))
            except Exception as e:
                # Skip files that can't be parsed
                continue

        print(f"  Found {config_classes_found} Config classes")
        print(f"  {config_classes_with_from_env} with from_env()")
        print(f"  {config_classes_with_from_cli} with from_cli()")

        # Check for YAML in executors
        executors_dir = self.repo_root / "executors"
        if executors_dir.exists():
            yaml_files = list(executors_dir.glob("**/*.yaml")) +
list(executors_dir.glob("**/*.yml"))
            if yaml_files:
                self.add_finding(Finding(
                    id=self.generate_finding_id("PARAM"),
                    severity=Severity.CRITICAL,
                    category="Parametrization",
                    title="YAML files prohibited in executors/",
                    description=f"Found {len(yaml_files)} YAML files in executors/
directory",
                    evidence=[str(f.relative_to(self.repo_root)) for f in yaml_files[:5]],
                    remediation="Remove all YAML files from executors/ and use Config
classes with from_env()/from_cli()",
                ))

    # ============================================================================
```

```
    # Phase 3: ArgRouter Validation
    # ============================================================================

    def audit_arg_router(self) -> None:
        """Verify ArgRouter with ≥30 routes and no silent drops."""
        print("📋 Phase 3: ArgRouter Validation")

        # Find ArgRouter implementation
        arg_router_files = list(self.src_root.glob("**/arg_router*.py"))

        if not arg_router_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("ARGROUTER"),
                severity=Severity.CRITICAL,
                category="ArgRouter",
                title="ArgRouter implementation not found",
                description="No arg_router*.py file found in source tree",
                remediation="Implement ArgRouter with typed route handling",
            ))
            return

        routes_count = 0
        has_kwargs_wildcard = False
        silent_drop_found = False

        for router_file in arg_router_files:
            content = router_file.read_text()
            tree = ast.parse(content)

            # Count route registrations and check for **kwargs
            for node in ast.walk(tree):
                if isinstance(node, ast.ClassDef) and "ArgRouter" in node.name:
                    # Count methods (approximate routes)
                    for method in node.body:
                        if isinstance(method, ast.FunctionDef):
                            if method.name not in ["__init__", "__repr__", "__str__"]:
                                routes_count += 1

                            # Check for **kwargs in signature
                            if method.args.kwarg:
                                has_kwargs_wildcard = True
                                self.add_finding(Finding(
                                    id=self.generate_finding_id("ARGROUTER"),
                                    severity=Severity.MEDIUM,
                                    category="ArgRouter",
                                    title=f"Method uses **kwargs: {method.name}",
                                    description="ArgRouter methods should have explicit
parameters, not **kwargs",

file_location=str(router_file.relative_to(self.repo_root)),
                                    line_number=method.lineno,
                                    remediation=f"Replace **kwargs with explicit
parameters in {method.name}",
                                ))

            # Check for silent drops
            if "silent" in content.lower() and "drop" in content.lower():
                silent_drop_found = True

        self.metrics.arg_router_routes_count = routes_count
        self.metrics.arg_router_silent_drops = 1 if silent_drop_found else 0

        print(f"  Found {routes_count} ArgRouter routes")

        if routes_count < 30:
            self.add_finding(Finding(
                id=self.generate_finding_id("ARGROUTER"),
                severity=Severity.MEDIUM,
```

```
                category="ArgRouter",
                title="Insufficient route count",
                description=f"Found {routes_count} routes, expected ≥30 specific routes",
                evidence=[f"Current routes: {routes_count}", "Expected: ≥30"],
                remediation="Add more specific routes to ArgRouter for all method types",
            ))

        if silent_drop_found:
            self.add_finding(Finding(
                id=self.generate_finding_id("ARGROUTER"),
                severity=Severity.HIGH,
                category="ArgRouter",
                title="Silent drop detected",
                description="ArgRouter contains silent drop logic",
                evidence=["Found 'silent' and 'drop' in code"],
                remediation="Remove silent drops and raise typed errors for all invalid
arguments",
            ))

    # ========================================================================
    # Phase 4: Cross-Cut Signals
    # ========================================================================

    def audit_signals(self) -> None:
        """Audit cross-cut signals system."""
        print("📡 Phase 4: Cross-Cut Signals")

        # Find signals implementation
        signals_files = list(self.src_root.glob("**/signals*.py"))

        if not signals_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("SIGNALS"),
                severity=Severity.HIGH,
                category="Signals",
                title="Signals implementation not found",
                description="No signals*.py file found in source tree",
                remediation="Implement cross-cut signal channel with SignalPack and
SignalRegistry",
            ))
            return

        for signals_file in signals_files:
            content = signals_file.read_text()

            # Check for memory:// support
            if "memory://" not in content:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SIGNALS"),
                    severity=Severity.MEDIUM,
                    category="Signals",
                    title="memory:// protocol not found",
                    description="Signals system should support memory:// for testing",
                    file_location=str(signals_file.relative_to(self.repo_root)),
                    remediation="Add memory:// protocol handler to SignalRegistry",
                ))

            # Check for HTTP client
            uses_http = "http://" in content or "https://" in content
            uses_httpx = "httpx" in content

            if uses_http and not uses_httpx:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SIGNALS"),
                    severity=Severity.HIGH,
                    category="Signals",
                    title="HTTP signals not using httpx",
                    description="HTTP signals should use httpx for async support",
```

```
                file_location=str(signals_file.relative_to(self.repo_root)),
                remediation="Replace HTTP client with httpx",
            ))

        if uses_http:
            # Check for circuit breaker
            has_circuit_breaker = "circuit" in content.lower() or "breaker" in
content.lower()
            has_etag = "etag" in content.lower()
            has_ttl = "ttl" in content.lower()

            if not has_circuit_breaker:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SIGNALS"),
                    severity=Severity.MEDIUM,
                    category="Signals",
                    title="HTTP signals missing circuit breaker",
                    description="HTTP signal client lacks circuit breaker pattern",
                    file_location=str(signals_file.relative_to(self.repo_root)),
                    remediation="Add circuit breaker with tenacity for HTTP
resilience",
                ))

            if not has_etag:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SIGNALS"),
                    severity=Severity.LOW,
                    category="Signals",
                    title="HTTP signals missing ETag support",
                    description="HTTP signals should support ETag for caching",
                    file_location=str(signals_file.relative_to(self.repo_root)),
                    remediation="Add ETag header handling for cache validation",
                ))

            if not has_ttl:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SIGNALS"),
                    severity=Severity.LOW,
                    category="Signals",
                    title="HTTP signals missing TTL",
                    description="HTTP signals should have explicit TTL",
                    file_location=str(signals_file.relative_to(self.repo_root)),
                    remediation="Add TTL configuration to signal cache",
                ))

        # Check for Pydantic validation
        if "pydantic" not in content.lower() and "BaseModel" not in content:
            self.add_finding(Finding(
                id=self.generate_finding_id("SIGNALS"),
                severity=Severity.MEDIUM,
                category="Signals",
                title="Signals missing Pydantic validation",
                description="SignalPack should use Pydantic for validation",
                file_location=str(signals_file.relative_to(self.repo_root)),
                remediation="Define SignalPack as Pydantic BaseModel",
            ))

    # ========================================================================
    # Phase 5: CPP→Orchestrator Validation
    # ========================================================================

    def audit_cpp_adapter(self) -> None:
        """Validate CPP adapter implementation."""
        print("♡ Phase 5: CPP→Orchestrator Validation")

        # Find CPP adapter
        cpp_adapter_files = list(self.src_root.glob("**/cpp_adapter*.py"))
```

```
        if not cpp_adapter_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("CPP"),
                severity=Severity.CRITICAL,
                category="CPP Adapter",
                title="CPP adapter not found",
                description="No cpp_adapter*.py file found",
                remediation="Implement CPPAdapter to convert CanonPolicyPackage to
PreprocessedDocument",
            ))
            return

        for adapter_file in cpp_adapter_files:
            content = adapter_file.read_text()

            # Check for ordering by text_span.start
            if "text_span.start" not in content:
                self.add_finding(Finding(
                    id=self.generate_finding_id("CPP"),
                    severity=Severity.HIGH,
                    category="CPP Adapter",
                    title="Missing text_span.start ordering",
                    description="CPP adapter should order chunks by text_span.start",
                    file_location=str(adapter_file.relative_to(self.repo_root)),
                    remediation="Add: chunks = sorted(chunks, key=lambda c:
c.text_span.start)",
                ))

            # Check for provenance_completeness
            if "provenance_completeness" not in content:
                self.add_finding(Finding(
                    id=self.generate_finding_id("CPP"),
                    severity=Severity.MEDIUM,
                    category="CPP Adapter",
                    title="Missing provenance_completeness calculation",
                    description="CPP adapter should compute provenance_completeness
metric",
                    file_location=str(adapter_file.relative_to(self.repo_root)),
                    remediation="Add provenance_completeness calculation (covered_span /
total_span)",
                ))

            # Check for ensure() method
            if "def ensure" not in content:
                self.add_finding(Finding(
                    id=self.generate_finding_id("CPP"),
                    severity=Severity.MEDIUM,
                    category="CPP Adapter",
                    title="Missing ensure() method",
                    description="CPP adapter should have ensure() for validation",
                    file_location=str(adapter_file.relative_to(self.repo_root)),
                    remediation="Add ensure() method for contract validation",
                ))

            # Check for resolution levels (micro/meso/macro)
            has_micro = "micro" in content.lower()
            has_meso = "meso" in content.lower()
            has_macro = "macro" in content.lower()

            if not (has_micro and has_meso and has_macro):
                missing = []
                if not has_micro:
                    missing.append("micro")
                if not has_meso:
                    missing.append("meso")
                if not has_macro:
                    missing.append("macro")
```

```
            self.add_finding(Finding(
                id=self.generate_finding_id("CPP"),
                severity=Severity.MEDIUM,
                category="CPP Adapter",
                title="Missing chunk resolution levels",
                description=f"CPP adapter missing resolution levels: {',
'.join(missing)}",
                file_location=str(adapter_file.relative_to(self.repo_root)),
                evidence=[f"Missing: {r}" for r in missing],
                remediation="Add support for all three resolution levels: micro, meso,
 macro",
            ))

    # ========================================================================
    # Phase 6: Determinism Check
    # ========================================================================

    def audit_determinism(self) -> None:
        """Check for deterministic execution."""
        print("🎲 Phase 6: Determinism Check")

        # Look for seed management
        seed_files = list(self.src_root.glob("**/seed*.py")) + \
                     list(self.src_root.glob("**/determinism/*.py"))

        if not seed_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("DETERM"),
                severity=Severity.HIGH,
                category="Determinism",
                title="No seed management found",
                description="No seed*.py or determinism/*.py files found",
                remediation="Implement seed factory for deterministic random number
generation",
            ))
        else:
            print(f"  Found {len(seed_files)} seed management files")

        # Check for random usage without seed
        py_files = list(self.src_root.glob("**/*.py"))
        unseeded_random_files = []

        for py_file in py_files:
            content = py_file.read_text()
            if "import random" in content or "from random import" in content:
                # Check if seed is set
                if "random.seed" not in content and "set_seed" not in content:
                    unseeded_random_files.append(py_file)

        if unseeded_random_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("DETERM"),
                severity=Severity.HIGH,
                category="Determinism",
                title="Random usage without seeding",
                description=f"Found {len(unseeded_random_files)} files using random
without seed",
                evidence=[str(f.relative_to(self.repo_root)) for f in
unseeded_random_files[:5]],
                remediation="Use seed_factory or call set_seed() before random
operations",
            ))

        # Check for phase_hash usage
        orchestrator_files = list(self.src_root.glob("**/orchestrator/**/*.py"))
        has_phase_hash = False

        for orc_file in orchestrator_files:
```

```
            content = orc_file.read_text()
            if "phase_hash" in content:
                has_phase_hash = True
                break

        if not has_phase_hash:
            self.add_finding(Finding(
                id=self.generate_finding_id("DETERM"),
                severity=Severity.MEDIUM,
                category="Determinism",
                title="phase_hash not found",
                description="Orchestrator should compute phase_hash for reproducibility
verification",
                remediation="Add phase_hash computation using blake3 of phase
inputs/outputs",
            ))

    # ========================================================================
    # Phase 7: Aggregation/Scoring Rules
    # ========================================================================

    def audit_aggregation_scoring(self) -> None:
        """Validate aggregation and scoring rules."""
        print("📋 Phase 7: Aggregation/Scoring Rules")

        # Find aggregation files
        agg_files = list(self.src_root.glob("**/aggregation*.py"))
        score_files = list(self.src_root.glob("**/scoring*.py"))

        if not agg_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("AGGREG"),
                severity=Severity.CRITICAL,
                category="Aggregation",
                title="Aggregation implementation not found",
                description="No aggregation*.py file found",
                remediation="Implement aggregation pipeline with explicit rules",
            ))
            return

        for agg_file in agg_files:
            content = agg_file.read_text()

            # Check for group_by keys
            if "group_by" not in content:
                self.add_finding(Finding(
                    id=self.generate_finding_id("AGGREG"),
                    severity=Severity.MEDIUM,
                    category="Aggregation",
                    title="Missing group_by specification",
                    description="Aggregation should have explicit group_by keys",
                    file_location=str(agg_file.relative_to(self.repo_root)),
                    remediation="Add explicit group_by parameter to aggregation
functions",
                ))

            # Check for weights
            if "weight" not in content.lower():
                self.add_finding(Finding(
                    id=self.generate_finding_id("AGGREG"),
                    severity=Severity.MEDIUM,
                    category="Aggregation",
                    title="Missing weight definitions",
                    description="Aggregation should have explicit weight definitions",
                    file_location=str(agg_file.relative_to(self.repo_root)),
                    remediation="Add weight configuration for aggregation rules",
                ))
```

```
            # Check for column validation
            has_validation = any(kw in content for kw in ["validate", "required",
"missing"])
            if not has_validation:
                self.add_finding(Finding(
                    id=self.generate_finding_id("AGGREG"),
                    severity=Severity.HIGH,
                    category="Aggregation",
                    title="Missing column validation",
                    description="Aggregation should fail on missing required columns",
                    file_location=str(agg_file.relative_to(self.repo_root)),
                    remediation="Add validation to raise error on missing required
columns",
                ))

    # ============================================================================
    # Phase 8: Reporting Artifacts
    # ============================================================================

    def audit_reporting(self) -> None:
        """Verify reporting artifacts."""
        print("📋 Phase 8: Reporting Artifacts")

        # Look for report generation
        report_files = list(self.src_root.glob("**/report*.py"))

        if not report_files:
            self.add_finding(Finding(
                id=self.generate_finding_id("REPORT"),
                severity=Severity.MEDIUM,
                category="Reporting",
                title="Report generation not found",
                description="No report*.py file found",
                remediation="Implement report generation with metrics and fingerprints",
            ))
            return

        for report_file in report_files:
            content = report_file.read_text()

            # Check for required report components
            components = {
                "metrics": "metric" in content.lower(),
                "fingerprints": "fingerprint" in content.lower(),
                "phase_hash": "phase_hash" in content,
                "used_signals": "used_signals" in content or "signal" in content.lower(),
            }

            missing = [k for k, v in components.items() if not v]
            if missing:
                self.add_finding(Finding(
                    id=self.generate_finding_id("REPORT"),
                    severity=Severity.MEDIUM,
                    category="Reporting",
                    title="Report missing required components",
                    description=f"Report lacks: {', '.join(missing)}",
                    file_location=str(report_file.relative_to(self.repo_root)),
                    evidence=[f"Missing: {c}" for c in missing],
                    remediation=f"Add {', '.join(missing)} to report generation",
                ))

    # ============================================================================
    # Phase 9: Security/Privacy
    # ============================================================================

    def audit_security_privacy(self) -> None:
        """Check security and privacy concerns."""
        print("📋 Phase 9: Security/Privacy")
```

```
        # Check for PII patterns in signals
        signals_files = list(self.src_root.glob("**/signals*.py"))

        pii_patterns = ["email", "phone", "ssn", "tax_id", "dni", "passport"]

        for signals_file in signals_files:
            content = signals_file.read_text().lower()
            found_pii = [p for p in pii_patterns if p in content]

            if found_pii:
                self.add_finding(Finding(
                    id=self.generate_finding_id("SECURITY"),
                    severity=Severity.HIGH,
                    category="Security/Privacy",
                    title="Potential PII in signals",
                    description=f"Found PII-related terms in signals: {',
'.join(found_pii)}",
                    file_location=str(signals_file.relative_to(self.repo_root)),
                    evidence=[f"Found: {p}" for p in found_pii],
                    remediation="Remove PII from signals or implement anonymization",
                ))

        # Check for hardcoded secrets
        py_files = list(self.src_root.glob("**/*.py"))
        secret_patterns = [
            r"password\s*=\s*['\"][^'\"]+['\"]",
            r"api_key\s*=\s*['\"][^'\"]+['\"]",
            r"secret\s*=\s*['\"][^'\"]+['\"]",
            r"token\s*=\s*['\"][^'\"]+['\"]",
        ]

        for py_file in py_files:
            content = py_file.read_text()
            for pattern in secret_patterns:
                matches = re.findall(pattern, content, re.IGNORECASE)
                if matches:
                    self.add_finding(Finding(
                        id=self.generate_finding_id("SECURITY"),
                        severity=Severity.CRITICAL,
                        category="Security/Privacy",
                        title="Hardcoded secret detected",
                        description=f"Found hardcoded secret in {py_file.name}",
                        file_location=str(py_file.relative_to(self.repo_root)),
                        evidence=matches[:2],
                        remediation="Move secrets to environment variables or secure
vault",
                    ))

        # Check HTTP client timeouts
        for py_file in py_files:
            content = py_file.read_text()
            if "httpx" in content or "requests" in content:
                if "timeout" not in content.lower():
                    self.add_finding(Finding(
                        id=self.generate_finding_id("SECURITY"),
                        severity=Severity.MEDIUM,
                        category="Security/Privacy",
                        title="HTTP client missing timeout",
                        description=f"HTTP client in {py_file.name} lacks timeout",
                        file_location=str(py_file.relative_to(self.repo_root)),
                        remediation="Add timeout parameter to all HTTP requests",
                    ))

    # ========================================================================
    # Phase 10: Dependencies
    # ========================================================================
```

```
def audit_dependencies(self) -> None:
    """Audit dependency management."""
    print("📦 Phase 10: Dependencies")

    # Check requirements files
    req_files = [
        self.repo_root / "requirements.txt",
        self.repo_root / "pyproject.toml",
    ]

    declared_deps = set()

    for req_file in req_files:
        if req_file.exists():
            content = req_file.read_text()
            # Extract package names (simple parsing)
            for line in content.split("\n"):
                line = line.strip()
                if line and not line.startswith("#"):
                    # Get package name before version specifier
                    pkg = re.split(r"[<>=!]", line)[0].strip()
                    if pkg:
                        declared_deps.add(pkg.lower())

    print(f"  Found {len(declared_deps)} declared dependencies")

    # Check for imports in source code
    imported_packages = set()
    py_files = list(self.src_root.glob("**/*.py"))

    for py_file in py_files:
        try:
            content = py_file.read_text()
            tree = ast.parse(content)

            for node in ast.walk(tree):
                if isinstance(node, ast.Import):
                    for alias in node.names:
                        pkg = alias.name.split(".")[0]
                        imported_packages.add(pkg.lower())
                elif isinstance(node, ast.ImportFrom):
                    if node.module:
                        pkg = node.module.split(".")[0]
                        imported_packages.add(pkg.lower())
        except:
            continue

    # Filter out stdlib and internal packages
    stdlib = {"os", "sys", "json", "re", "time", "datetime", "pathlib", "typing",
              "dataclasses", "collections", "itertools", "functools", "abc",
              "logging", "threading", "asyncio", "inspect", "ast", "hashlib"}

    external_imports = imported_packages - stdlib - {"saaaaaa"}

    # Find undeclared dependencies
    undeclared = external_imports - declared_deps

    if undeclared:
        # Filter common false positives
        undeclared = {pkg for pkg in undeclared if len(pkg) > 2}

        if undeclared:
            self.add_finding(Finding(
                id=self.generate_finding_id("DEPS"),
                severity=Severity.HIGH,
                category="Dependencies",
                title="Undeclared dependencies detected",
                description=f"Found {len(undeclared)} imported packages not in
```

```
requirements",
                evidence=sorted(list(undeclared))[:10],
                remediation="Add missing packages to requirements.txt with version
pins",
            ))

        # Check for version pins
        req_file = self.repo_root / "requirements.txt"
        if req_file.exists():
            content = req_file.read_text()
            unpinned = []
            for line in content.split("\n"):
                line = line.strip()
                if line and not line.startswith("#"):
                    if "==" not in line:
                        unpinned.append(line)

            if unpinned:
                self.add_finding(Finding(
                    id=self.generate_finding_id("DEPS"),
                    severity=Severity.MEDIUM,
                    category="Dependencies",
                    title="Unpinned dependencies",
                    description=f"Found {len(unpinned)} dependencies without exact version
 pins",
                    evidence=unpinned[:5],
                    remediation="Pin all dependencies with == to ensure reproducibility",
                ))

    # ========================================================================
    # Main Audit Execution
    # ========================================================================

    def run_audit(self) -> None:
        """Execute all audit phases."""
        print("\n" + "=" * 80)
        print("COMPREHENSIVE PIPELINE TECHNICAL AUDIT")
        print("=" * 80)
        print(f"Repository: {self.repo_root}")
        print(f"Timestamp: {datetime.now().isoformat()}")
        print("=" * 80 + "\n")

        # Run all audit phases
        self.audit_contract_compatibility()
        self.audit_parametrization()
        self.audit_arg_router()
        self.audit_signals()
        self.audit_cpp_adapter()
        self.audit_determinism()
        self.audit_aggregation_scoring()
        self.audit_reporting()
        self.audit_security_privacy()
        self.audit_dependencies()

        print("\n" + "=" * 80)
        print("AUDIT COMPLETE")
        print("=" * 80)

    def generate_reports(self) -> Tuple[int, int]:
        """Generate audit reports and return critical/total counts."""
        # Count findings by severity
        severity_counts = defaultdict(int)
        for finding in self.findings:
            severity_counts[finding.severity] += 1

        critical_count = severity_counts[Severity.CRITICAL]
        high_count = severity_counts[Severity.HIGH]
        medium_count = severity_counts[Severity.MEDIUM]
```

```
    low_count = severity_counts[Severity.LOW]
    info_count = severity_counts[Severity.INFO]
    total_count = len(self.findings)

    # Generate AUDIT_REPORT.md
    self._generate_audit_report(
        severity_counts, critical_count, high_count,
        medium_count, low_count, total_count
    )

    # Generate AUDIT_FIX_PLAN.md
    self._generate_fix_plan(critical_count, high_count, medium_count)

    return critical_count, total_count

def _generate_audit_report(
    self,
    severity_counts: Dict[Severity, int],
    critical_count: int,
    high_count: int,
    medium_count: int,
    low_count: int,
    total_count: int
) -> None:
    """Generate AUDIT_REPORT.md."""
    report_path = self.repo_root / "AUDIT_REPORT.md"

    with open(report_path, "w") as f:
        f.write("# Comprehensive Pipeline Technical Audit Report\n\n")
        f.write(f"**Generated:** {datetime.now().isoformat()}\n\n")
        f.write(f"**Repository:** {self.repo_root}\n\n")

        # Executive Summary
        f.write("## Executive Summary\n\n")
        f.write(f"**Total Findings:** {total_count}\n\n")
        f.write("| Severity | Count |\n")
        f.write("|----------|-------|\n")
        f.write(f"| 🔴 CRITICAL | {critical_count} |\n")
        f.write(f"| 🟠 HIGH | {high_count} |\n")
        f.write(f"| 🟡 MEDIUM | {medium_count} |\n")
        f.write(f"| 🟢 LOW | {low_count} |\n")
        f.write(f"| ℹ️ INFO | {severity_counts[Severity.INFO]} |\n\n")

        # Audit Metrics
        f.write("## Audit Metrics\n\n")
        f.write("```json\n")
        f.write(json.dumps(self.metrics.to_dict(), indent=2))
        f.write("\n```\n\n")

        # Contract Matrix
        f.write("## Contract Compatibility Matrix\n\n")
        f.write("| Stage | Input Contract | Output Contract | Status |\n")
        f.write("|-------|---------------|-----------------|--------|\n")
        f.write("| Ingest | Document | PreprocessedDocument | ⚠️ |\n")
        f.write("| Normalize | PreprocessedDocument | CanonPolicyPackage | ⚠️ |\n")
        f.write("| Chunk | CanonPolicyPackage | ChunkGraph | ⚠️ |\n")
        f.write("| Signals | - | SignalPack | ⚠️ |\n")
        f.write("| Aggregate | ScoredResult[] | AreaScore | ⚠️ |\n")
        f.write("| Score | AreaScore | MacroScore | ⚠️ |\n")
        f.write("| Report | MacroScore | Report | ⚠️ |\n\n")

        # Findings by Category
        f.write("## Findings by Category\n\n")

        findings_by_category = defaultdict(list)
        for finding in self.findings:
            findings_by_category[finding.category].append(finding)
```

```
            for category in sorted(findings_by_category.keys()):
                findings = findings_by_category[category]
                f.write(f"### {category} ({len(findings)} findings)\n\n")

                for finding in sorted(findings, key=lambda x: x.severity.value):
                    severity_emoji = {
                        Severity.CRITICAL: "🔴",
                        Severity.HIGH: "🟠",
                        Severity.MEDIUM: "🟡",
                        Severity.LOW: "🟢",
                        Severity.INFO: "ℹ️",
                    }[finding.severity]

                    f.write(f"#### {severity_emoji} {finding.id}: {finding.title}\n\n")
                    f.write(f"**Severity:** {finding.severity.value}\n\n")
                    f.write(f"**Description:** {finding.description}\n\n")

                    if finding.file_location:
                        location = finding.file_location
                        if finding.line_number:
                            location += f":{finding.line_number}"
                        f.write(f"**Location:** `{location}`\n\n")

                    if finding.evidence:
                        f.write("**Evidence:**\n")
                        for evidence in finding.evidence:
                            f.write(f"- {evidence}\n")
                        f.write("\n")

                    if finding.remediation:
                        f.write(f"**Remediation:** {finding.remediation}\n\n")

                    f.write("---\n\n")

            # Summary
            f.write("## Summary\n\n")
            if critical_count > 0:
                f.write(f"⚠️ **{critical_count} CRITICAL findings require immediate
attention**\n\n")
            if high_count > 0:
                f.write(f"⚠️ {high_count} HIGH priority findings should be addressed
soon\n\n")
            if medium_count + low_count > 0:
                f.write(f"ℹ️ {medium_count + low_count} MEDIUM/LOW priority findings for
improvement\n\n")

        print(f"\n✓ Generated: {report_path}")

    def _generate_fix_plan(
        self,
        critical_count: int,
        high_count: int,
        medium_count: int
    ) -> None:
        """Generate AUDIT_FIX_PLAN.md."""
        plan_path = self.repo_root / "AUDIT_FIX_PLAN.md"

        with open(plan_path, "w") as f:
            f.write("# Audit Fix Plan\n\n")
            f.write(f"**Generated:** {datetime.now().isoformat()}\n\n")

            # Immediate (Critical)
            critical_findings = [f for f in self.findings if f.severity ==
Severity.CRITICAL]
            if critical_findings:
                f.write("## Immediate Priority (CRITICAL)\n\n")
                f.write("**Timeline:** Within 24 hours\n\n")
                f.write("**Owner:** Development Team Lead\n\n")
```

```
            for finding in critical_findings:
                f.write(f"### {finding.id}: {finding.title}\n\n")
                f.write(f"**Issue:** {finding.description}\n\n")
                if finding.remediation:
                    f.write(f"**Action:** {finding.remediation}\n\n")
                if finding.file_location:
                    f.write(f"**File:** `{finding.file_location}`\n\n")
                f.write("---\n\n")

        # Short-Term (High)
        high_findings = [f for f in self.findings if f.severity == Severity.HIGH]
        if high_findings:
            f.write("## Short-Term Priority (HIGH)\n\n")
            f.write("**Timeline:** Within 1 week\n\n")
            f.write("**Owner:** Development Team\n\n")

            for finding in high_findings:
                f.write(f"### {finding.id}: {finding.title}\n\n")
                f.write(f"**Issue:** {finding.description}\n\n")
                if finding.remediation:
                    f.write(f"**Action:** {finding.remediation}\n\n")
                if finding.file_location:
                    f.write(f"**File:** `{finding.file_location}`\n\n")
                f.write("---\n\n")

        # Medium-Term (Medium)
        medium_findings = [f for f in self.findings if f.severity == Severity.MEDIUM]
        if medium_findings:
            f.write("## Medium-Term Priority (MEDIUM)\n\n")
            f.write("**Timeline:** Within 2-4 weeks\n\n")
            f.write("**Owner:** Development Team\n\n")

            for finding in medium_findings[:10]:  # Limit to first 10
                f.write(f"### {finding.id}: {finding.title}\n\n")
                f.write(f"**Issue:** {finding.description}\n\n")
                if finding.remediation:
                    f.write(f"**Action:** {finding.remediation}\n\n")
                f.write("---\n\n")

            if len(medium_findings) > 10:
                f.write(f"*... and {len(medium_findings) - 10} more medium priority
items*\n\n")

        print(f"✓ Generated: {plan_path}")


def main() -> int:
    """Main entry point."""
    repo_root = Path(__file__).parent

    auditor = ComprehensivePipelineAuditor(repo_root)
    auditor.run_audit()

    critical_count, total_count = auditor.generate_reports()

    print(f"\n{'=' * 80}")
    print(f"Total Findings: {total_count}")
    print(f"Critical Findings: {critical_count}")
    print(f"{'=' * 80}\n")

    # Exit with non-zero if critical findings exist
    return 1 if critical_count > 0 else 0


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

```
===== FILE: scripts/count_producer_methods.py =====
#!/usr/bin/env python3
"""Count methods in Producer classes"""
import ast
import json
from pathlib import Path

def count_methods_in_class(filepath: Path, class_name: str) -> dict[str, int]:
    """Count public and private methods in a class"""
    with open(filepath, encoding='utf-8') as f:
        tree = ast.parse(f.read())

    method_counts = {
        "public": 0,
        "private": 0,
        "total": 0
    }

    for node in ast.walk(tree):
        if isinstance(node, ast.ClassDef) and node.name == class_name:
            for item in node.body:
                if isinstance(item, ast.FunctionDef):
                    if not item.name.startswith('_'):
                        method_counts["public"] += 1
                    else:
                        method_counts["private"] += 1
                    method_counts["total"] += 1

    return method_counts

# Count methods in each Producer
producers = {
    "SemanticChunkingProducer": "semantic_chunking_policy.py",
    "EmbeddingPolicyProducer": "embedding_policy.py",
    "DerekBeachProducer": "derek_beach.py",
    "ReportAssemblyProducer": "report_assembly.py"
}

results = {}
total_public = 0

for class_name, filepath in producers.items():
    counts = count_methods_in_class(Path(filepath), class_name)
    results[class_name] = counts
    total_public += counts["public"]
    print(f"{class_name}: {counts['public']} public methods, {counts['private']} private,
{counts['total']} total")

print(f"\nTotal public methods across all producers: {total_public}")

# Save results
with open('method_counts.json', 'w') as f:
    json.dump({
        "producers": results,
        "total_public_methods": total_public
    }, f, indent=2)

print("\nResults saved to method_counts.json")

===== FILE: scripts/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',
}

def should_include(path: Path, base: Path) -> bool:
    """Determine if a file should be included in the deployment zip."""
    relative_path = path.relative_to(base)
    path_str = str(relative_path)

    # Check if it's the deprecated ORCHESTRATOR_MONILITH
    if 'ORCHESTRATOR_MONILITH.py' in path_str:
        return False

    # Exclude markdown files except essential ones
    if path_str.endswith('.md'):
```

```
        return path_str in ESSENTIAL_DOCS

    # Check exclude patterns
    for pattern in EXCLUDE_PATTERNS:
        if pattern.endswith('/'):
            # Directory pattern - match at start of path components
            pattern_name = pattern.rstrip('/')
            if path_str.startswith(pattern_name + '/') or path_str == pattern_name:
                return False
        elif pattern.startswith('*.'):
            # Extension pattern
            if path_str.endswith(pattern[1:]):
                return False
        elif '/' in pattern:
            # Path pattern - exact match
            if path_str == pattern or path_str.startswith(pattern + '/'):
                return False
        else:
            # Filename pattern - match exact filename component
            parts = Path(path_str).parts
            if pattern in parts:
                return False

    # Check include patterns
    for pattern in INCLUDE_PATTERNS:
        if pattern.endswith('/'):
            # Directory pattern
            if path_str.startswith(pattern.rstrip('/')):
                return True
        elif pattern == path_str:
            # Exact match
            return True

    # If in src/ directory, include by default
    return bool(path_str.startswith('src/'))

def create_deployment_zip(output_path: Path) -> None:
    """Create a deployment zip file."""
    print(f"Creating deployment zip at {output_path}")

    included_files: list[str] = []
    excluded_files: list[str] = []

    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("\n✓ 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("\n📦 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("\n🚫 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"\n📄 Manifest saved to: {manifest_path}")

if __name__ == "__main__":
    output_zip = BASE_DIR / "saaaaaa-deployment.zip"
    create_deployment_zip(output_zip)
    print("\n✨ Deployment package ready for production!")

===== FILE: scripts/equip_compat.py =====
#!/usr/bin/env python3
"""
Compatibility Layer Equipment Script

Verifies the compat layer functionality including:
- safe_imports module
- native_check module
- Version compatibility shims
- Platform detection

Exit codes:
- 0: Compat layer OK
- 1: Compat layer has issues
"""

from __future__ import annotations

import sys
from pathlib import Path
```

```
def test_compat_imports() -> bool:
    """Test that compat module can be imported."""
    print("=== Compat Module Imports ===")

    try:
        from saaaaaa.compat import (
            ImportErrorDetailed,
            check_import_available,
            get_import_version,
            lazy_import,
            tomllib,
            try_import,
        )
        print("✓ All compat exports available")
        return True
    except ImportError as e:
        print(f"✗ Failed to import compat: {e}")
        return False


def test_safe_imports_functionality() -> bool:
    """Test safe_imports functions."""
    print("\n=== Safe Imports Functionality ===")

    from saaaaaa.compat import check_import_available, try_import

    all_ok = True

    # Test checking for existing module
    if check_import_available("sys"):
        print("✓ check_import_available works for stdlib")
    else:
        print("✗ check_import_available failed for sys")
        all_ok = False

    # Test importing existing module
    result = try_import("os", required=False)
    if result is not None:
        print("✓ try_import works for stdlib")
    else:
        print("✗ try_import failed for os")
        all_ok = False

    # Test importing nonexistent optional module (should not raise)
    result = try_import("nonexistent_test_module", required=False)
    if result is None:
        print("✓ try_import handles missing optional correctly")
    else:
        print("✗ try_import should return None for missing optional")
        all_ok = False

    return all_ok


def test_native_check() -> bool:
    """Test native_check module."""
    print("\n=== Native Check Functionality ===")

    try:
        from saaaaaa.compat.native_check import (
            check_cpu_features,
            check_system_library,
        )

        # Test CPU features
        cpu_result = check_cpu_features()
```

```
        print(f"✓ CPU features check: {cpu_result.message}")

        # Test system library check (informational)
        lib_result = check_system_library("zstd")
        status = "available" if lib_result.available else "not found"
        print(f"  System library zstd: {status}")

        return True
    except Exception as e:
        print(f"✗ Native check failed: {e}")
        return False


def test_version_shims() -> bool:
    """Test version compatibility shims."""
    print("\n=== Version Compatibility Shims ===")

    from saaaaaa.compat import tomllib

    all_ok = True

    # Test tomllib
    if tomllib is not None:
        print("✓ TOML support available (tomllib or tomli)")
    else:
        print("✗ TOML support not available")
        all_ok = False

    # Test typing extensions
    try:
        from saaaaaa.compat import (
            Annotated,
            Final,
            Literal,
            Protocol,
            TypeAlias,
            TypedDict,
        )
        print("✓ Typing extensions available")
    except ImportError as e:
        print(f"✗ Typing extensions failed: {e}")
        all_ok = False

    return all_ok


def main() -> int:
    """Main entry point."""
    print("=" * 60)
    print("COMPATIBILITY LAYER EQUIPMENT CHECK")
    print("=" * 60)
    print()

    checks = [
        ("Compat Imports", test_compat_imports()),
        ("Safe Imports", test_safe_imports_functionality()),
        ("Native Check", test_native_check()),
        ("Version Shims", test_version_shims()),
    ]

    # Summary
    print("\n" + "=" * 60)
    print("SUMMARY")
    print("=" * 60)

    failed = []
    for name, passed in checks:
        status = "✓" if passed else "✗"
```

```
        print(f"{status} {name}")
        if not passed:
            failed.append(name)

    print()

    if failed:
        print(f"Failed checks: {', '.join(failed)}")
        return 1
    else:
        print("✓ Compat layer is ready!")
        return 0


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

===== FILE: scripts/equip_cpp_smoke.py =====
#!/usr/bin/env python3
"""
Equipment script for CPP subsystem.

Runs smoke tests for SPCAdapter and CPPIngestionPipeline.
"""

import sys
import traceback
from pathlib import Path
from typing import Dict, Any


def test_cpp_adapter_import() -> Dict[str, Any]:
    """Test SPCAdapter can be imported."""
    try:
        from saaaaaa.utils.spc_adapter import SPCAdapter, adapt_spc_to_orchestrator
        return {
            "success": True,
            "message": "SPCAdapter importable"
        }
    except ImportError as e:
        return {
            "success": False,
            "message": f"Import failed: {e}"
        }


def test_cpp_ingestion_pipeline() -> Dict[str, Any]:
    """Test CPPIngestionPipeline initialization."""
    try:
        from saaaaaa.processing.cpp_ingestion import CPPIngestionPipeline

        pipeline = CPPIngestionPipeline(
            enable_ocr=False,
            ocr_confidence_threshold=0.85,
            chunk_overlap_threshold=0.15
        )

        return {
            "success": True,
            "schema_version": pipeline.SCHEMA_VERSION,
            "message": f"CPPIngestionPipeline initialized
(schema={pipeline.SCHEMA_VERSION})"
        }
    except Exception as e:
        return {
            "success": False,
            "message": f"Initialization failed: {e}"
        }
```

```
def test_cpp_adapter_conversion() -> Dict[str, Any]:
    """Test SPCAdapter conversion with minimal CPP document."""
    try:
        from saaaaaa.utils.spc_adapter import SPCAdapter
        from saaaaaa.processing.cpp_ingestion.models import (
            CanonPolicyPackage,
            ChunkGraph,
            Chunk,
            ChunkResolution,
            TextSpan,
            PolicyManifest,
            ProvenanceMap,
            QualityMetrics,
            IntegrityIndex,
        )

        # Create minimal test CPP
        chunk = Chunk(
            id="test_chunk_001",
            bytes_hash="test_hash",
            text_span=TextSpan(start=0, end=100),
            resolution=ChunkResolution.MICRO,
            text="Test policy document text.",
            policy_facets=None,
            time_facets=None,
            geo_facets=None,
        )

        chunk_graph = ChunkGraph()
        chunk_graph.add_chunk(chunk)

        policy_manifest = PolicyManifest(
            axes=["test_axis"],
            programs=["test_program"],
            years=[2024],
            territories=["test_territory"]
        )

        provenance_map = ProvenanceMap(
            source_document="test_doc.pdf",
            ingestion_timestamp="2025-11-06T00:00:00Z",
            pipeline_version="1.0.0"
        )

        quality_metrics = QualityMetrics(
            boundary_f1=0.95,
            kpi_linkage_rate=0.90,
            budget_consistency_score=0.85,
            provenance_completeness=1.0
        )

        integrity_index = IntegrityIndex(
            chunk_count=1,
            total_bytes=100,
            global_hash="test_global_hash"
        )

        cpp = CanonPolicyPackage(
            chunk_graph=chunk_graph,
            policy_manifest=policy_manifest,
            provenance_map=provenance_map,
            quality_metrics=quality_metrics,
            integrity_index=integrity_index,
            schema_version="1.0.0"
        )
```

```
        # Test conversion
        adapter = SPCAdapter()
        preprocessed = adapter.adapt(cpp)

        return {
            "success": True,
            "provenance_completeness": cpp.quality_metrics.provenance_completeness,
            "chunk_count": len(preprocessed.sentences),
            "message": f"Conversion successful
(provenance={cpp.quality_metrics.provenance_completeness})"
        }
    except Exception as e:
        return {
            "success": False,
            "message": f"Conversion failed: {e}",
            "traceback": traceback.format_exc()
        }


def test_cpp_ensure() -> Dict[str, Any]:
    """Test SPCAdapter.ensure() method."""
    try:
        from saaaaaa.utils.spc_adapter import SPCAdapter
        from saaaaaa.processing.cpp_ingestion.models import CanonPolicyPackage

        # Create adapter
        adapter = SPCAdapter()

        # Test with None (should raise)
        try:
            adapter.ensure(None)
            return {
                "success": False,
                "message": "ensure(None) should raise SPCAdapterError"
            }
        except Exception:
            pass  # Expected

        return {
            "success": True,
            "message": "ensure() validation working"
        }
    except Exception as e:
        return {
            "success": False,
            "message": f"ensure() test failed: {e}"
        }


def main():
    """Run CPP equipment smoke tests."""
    print("=" * 70)
    print("EQUIP:CPP - CPP Adapter & Ingestion")
    print("=" * 70)
    print()

    tests = [
        ("SPCAdapter import", test_cpp_adapter_import),
        ("CPPIngestionPipeline init", test_cpp_ingestion_pipeline),
        ("SPCAdapter conversion", test_cpp_adapter_conversion),
        ("SPCAdapter ensure()", test_cpp_ensure),
    ]

    results = []
    for name, test_func in tests:
        print(f"Testing: {name}...")
        result = test_func()
        results.append(result['success'])
```

```
        if result['success']:
            print(f"✓ {result['message']}")
        else:
            print(f"✗ {result['message']}")
            if 'traceback' in result:
                print(f"  Traceback:\n{result['traceback']}")
        print()

    print("=" * 70)
    if all(results):
        print(f"✓ CPP EQUIPMENT COMPLETE: {len(results)}/{len(results)} tests passed")
    else:
        failed = sum(1 for r in results if not r)
        print(f"✗ CPP EQUIPMENT FAILED: {failed}/{len(results)} tests failed")
    print("=" * 70)

    return 0 if all(results) else 1


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

===== FILE: scripts/equip_native.py =====
#!/usr/bin/env python3
"""
Native Dependencies Equipment Script

Verifies system libraries and native extensions for:
- C-extensions (pyarrow, polars, blake3)
- System libraries (zstd, icu, omp)
- Platform compatibility
- CPU features

Exit codes:
- 0: All native dependencies OK (or warnings only)
- 1: Critical native dependencies missing
"""

from __future__ import annotations

import sys
from pathlib import Path

from saaaaaa.compat.native_check import (
    check_cpu_features,
    check_fips_mode,
    check_system_library,
    verify_native_dependencies,
)


def main() -> int:
    """Main entry point."""
    print("=" * 60)
    print("NATIVE DEPENDENCIES EQUIPMENT CHECK")
    print("=" * 60)
    print()

    # Use the comprehensive report from native_check
    from saaaaaa.compat.native_check import print_native_report
    print_native_report()

    print("=" * 60)
    print("RECOMMENDATION")
    print("=" * 60)

    # Check critical packages
```

```
    critical = ["pyarrow", "blake3"]
    results = verify_native_dependencies(critical)

    missing_critical = [
        name for name, result in results.items()
        if not result.available and ":" not in name
    ]

    if missing_critical:
        print(f"✗ Missing critical native packages: {', '.join(missing_critical)}")
        print("Install with: pip install -r requirements.txt")
        return 1
    else:
        print("✓ All critical native dependencies available")
        print("\nNote: Some system libraries may be missing but are not critical")
        print("      for basic functionality. See warnings above.")
        return 0


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

===== FILE: scripts/equip_python.py =====
#!/usr/bin/env python3
"""
Python Environment Equipment Script

Verifies Python environment readiness including:
- Python version requirements
- Package dependencies
- C-extensions compilation
- Import availability for critical packages

Exit codes:
- 0: Environment ready
- 1: Environment has issues
"""

from __future__ import annotations

import importlib
import subprocess
import sys
from pathlib import Path


def check_python_version() -> bool:
    """Check Python version meets minimum requirements."""
    print("=== Python Version Check ===")
    min_version = (3, 10)
    current = sys.version_info[:2]

    print(f"Current: Python {current[0]}.{current[1]}")
    print(f"Required: Python {min_version[0]}.{min_version[1]}+")

    if current >= min_version:
        print("✓ Version OK\n")
        return True
    else:
        print(f"✗ Python {min_version[0]}.{min_version[1]}+ required\n")
        return False


def check_critical_imports() -> bool:
    """Check that critical packages can be imported."""
    print("=== Critical Package Imports ===")

    critical_packages = [
```

```
        ("numpy", "Core scientific computing"),
        ("pandas", "Data manipulation"),
        ("pydantic", "Data validation"),
        ("blake3", "Cryptographic hashing"),
        ("structlog", "Structured logging"),
    ]

    all_ok = True
    for package, description in critical_packages:
        try:
            importlib.import_module(package)
            print(f"✓ {package}: {description}")
        except ImportError as e:
            print(f"✗ {package}: NOT INSTALLED ({description})")
            all_ok = False

    print()
    return all_ok


def check_optional_imports() -> None:
    """Check optional packages (informational only)."""
    print("=== Optional Package Imports (Informational) ===")

    optional_packages = [
        ("polars", "Fast DataFrame library"),
        ("pyarrow", "Arrow format support"),
        ("torch", "Deep learning"),
        ("tensorflow", "Machine learning"),
        ("transformers", "NLP models"),
        ("spacy", "NLP processing"),
    ]

    for package, description in optional_packages:
        try:
            importlib.import_module(package)
            print(f"✓ {package}: {description}")
        except ImportError:
            print(f"  {package}: not installed ({description})")

    print()


def test_package_import() -> bool:
    """Test that the saaaaaa package can be imported."""
    print("=== SAAAAAA Package Import ===")

    try:
        import saaaaaa
        print(f"✓ Package imported successfully")

        # Test compat layer
        from saaaaaa.compat import try_import
        print(f"✓ Compat layer available")

        print()
        return True
    except Exception as e:
        print(f"✗ Failed to import package: {e}\n")
        return False


def compile_bytecode() -> bool:
    """Compile Python bytecode to check for syntax errors."""
    print("=== Bytecode Compilation ===")

    root = Path(__file__).parent.parent
    src_path = root / "src" / "saaaaaa"
```

```
    try:
        result = subprocess.run(
            ["python", "-m", "compileall", "-q", str(src_path)],
            capture_output=True,
            text=True,
            timeout=30,
        )

        if result.returncode == 0:
            print("✓ All files compile successfully\n")
            return True
        else:
            print(f"✗ Compilation errors:\n{result.stderr}\n")
            return False
    except Exception as e:
        print(f"✗ Compilation check failed: {e}\n")
        return False


def main() -> int:
    """Main entry point."""
    print("=" * 60)
    print("PYTHON ENVIRONMENT EQUIPMENT CHECK")
    print("=" * 60)
    print()

    checks = [
        ("Python Version", check_python_version()),
        ("Critical Imports", check_critical_imports()),
        ("Package Import", test_package_import()),
        ("Bytecode Compilation", compile_bytecode()),
    ]

    # Run optional checks (don't affect exit code)
    check_optional_imports()

    # Summary
    print("=" * 60)
    print("SUMMARY")
    print("=" * 60)

    failed = []
    for name, passed in checks:
        status = "✓" if passed else "✗"
        print(f"{status} {name}")
        if not passed:
            failed.append(name)

    print()

    if failed:
        print(f"Failed checks: {', '.join(failed)}")
        print("Please resolve these issues before proceeding.")
        return 1
    else:
        print("✓ Environment is ready!")
        return 0


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

===== FILE: scripts/equip_signals.py =====
#!/usr/bin/env python3
"""
Equipment script for signals subsystem.
```

```
Initializes SignalRegistry, warms up memory cache, and verifies hit rates.
"""

import argparse
import sys
from typing import Dict, Any


def warmup_memory_signals() -> Dict[str, Any]:
    """Warm up memory:// signal cache with test data."""
    from saaaaaa.core.orchestrator.signals import SignalClient, SignalPack

    client = SignalClient(base_url="memory://")

    # Register test signals for common policy areas
    policy_areas = [
        "fiscal", "education", "health", "infrastructure", "security",
        "environment", "social", "economic", "governance", "culture"
    ]

    registered = 0
    for area in policy_areas:
        signal_pack = SignalPack(
            version="1.0.0",
            policy_area=area,
            patterns=[f"pattern_{area}_1", f"pattern_{area}_2", f"pattern_{area}_3"],
            indicators=[f"indicator_{area}"],
            regex=[f"regex_{area}"],
            verbs=[f"verb_{area}"],
            entities=[f"entity_{area}"],
            thresholds={f"threshold_{area}": 0.85}
        )
        client.register_memory_signal(area, signal_pack)
        registered += 1

    return {
        "registered": registered,
        "policy_areas": policy_areas,
        "client_base_url": client.base_url
    }


def initialize_signal_registry(max_size: int = 100, ttl_s: int = 3600) -> Dict[str, Any]:
    """Initialize SignalRegistry with specified parameters."""
    from saaaaaa.core.orchestrator.signals import SignalRegistry

    registry = SignalRegistry(max_size=max_size, default_ttl_s=ttl_s)

    return {
        "max_size": registry._max_size,
        "default_ttl_s": registry._default_ttl_s,
        "store_size": len(registry._store)
    }


def verify_signal_hit_rate(threshold: float = 0.95) -> Dict[str, Any]:
    """Verify signal hit rate meets threshold."""
    from saaaaaa.core.orchestrator.signals import SignalClient

    client = SignalClient(base_url="memory://")

    # Test fetching registered signals
    test_areas = ["fiscal", "education", "health"]
    hits = 0
    total = len(test_areas)

    for area in test_areas:
        signal_pack = client.fetch_signal_pack(area)
```

```
        if signal_pack is not None:
            hits += 1

    hit_rate = hits / total if total > 0 else 0.0
    passed = hit_rate >= threshold

    return {
        "hits": hits,
        "total": total,
        "hit_rate": hit_rate,
        "threshold": threshold,
        "passed": passed
    }


def precompile_patterns() -> Dict[str, Any]:
    """Pre-compile common regex patterns."""
    import re

    patterns = [
        r"\d+\.\d+",  # Decimal numbers
        r"\$\s*\d+(?:,\d{3})*(?:\.\d{2})?",  # Currency amounts
        r"\d{4}-\d{4}",  # Year ranges
        r"(?:Ley|Decreto|Resolución)\s+\d+",  # Legal references
    ]

    compiled = []
    for pattern in patterns:
        try:
            re.compile(pattern)
            compiled.append(pattern)
        except re.error:
            pass

    return {
        "total_patterns": len(patterns),
        "compiled": len(compiled),
        "patterns": compiled
    }


def main():
    """Main equipment routine for signals."""
    parser = argparse.ArgumentParser(
        description="Equipment routine for signals subsystem"
    )
    parser.add_argument(
        "--source",
        default="memory",
        choices=["memory", "http"],
        help="Signal source (default: memory)"
    )
    parser.add_argument(
        "--preload-patterns",
        action="store_true",
        help="Pre-compile regex patterns"
    )
    parser.add_argument(
        "--warmup-cache",
        action="store_true",
        help="Warm up signal cache"
    )
    parser.add_argument(
        "--verify-registry",
        action="store_true",
        help="Verify signal registry initialization"
    )
    parser.add_argument(
```

```
        "--hit-rate-threshold",
        type=float,
        default=0.95,
        help="Minimum hit rate threshold (default: 0.95)"
    )

    args = parser.parse_args()

    print("=" * 70)
    print("EQUIP:SIGNALS - Sistema de Señales")
    print("=" * 70)
    print()

    all_passed = True

    # Initialize registry
    if args.verify_registry:
        print("Inicializando SignalRegistry...")
        try:
            result = initialize_signal_registry()
            print(f"✓ SignalRegistry: max_size={result['max_size']},
ttl={result['default_ttl_s']}s")
        except Exception as e:
            print(f"✗ SignalRegistry initialization failed: {e}")
            all_passed = False

    # Warm up cache
    if args.warmup_cache:
        print("\nPre-calentamiento de cache...")
        try:
            result = warmup_memory_signals()
            print(f"✓ Cache warmed: {result['registered']} policy areas registered")
            print(f"  Areas: {', '.join(result['policy_areas'][:5])}...")
        except Exception as e:
            print(f"✗ Cache warmup failed: {e}")
            all_passed = False

    # Pre-compile patterns
    if args.preload_patterns:
        print("\nPre-compilando patrones regex...")
        try:
            result = precompile_patterns()
            print(f"✓ Patterns compiled: {result['compiled']}/{result['total_patterns']}")
        except Exception as e:
            print(f"✗ Pattern compilation failed: {e}")
            all_passed = False

    # Verify hit rate
    print("\nVerificando hit rate de señales...")
    try:
        result = verify_signal_hit_rate(args.hit_rate_threshold)
        if result['passed']:
            print(f"✓ Hit rate: {result['hit_rate']:.1%} (threshold:
{result['threshold']:.1%})")
        else:
            print(f"✗ Hit rate: {result['hit_rate']:.1%} < {result['threshold']:.1%}")
            all_passed = False
    except Exception as e:
        print(f"✗ Hit rate verification failed: {e}")
        all_passed = False

    print()
    print("=" * 70)
    if all_passed:
        print("✓ SIGNALS EQUIPMENT COMPLETE")
    else:
        print("✗ SIGNALS EQUIPMENT FAILED")
    print("=" * 70)
```

```
    return 0 if all_passed else 1


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

===== FILE: scripts/execute_deletion.py =====
#!/usr/bin/env python3
"""
PHASE 1: MASSIVE DELETION - EXECUTION SCRIPT
Executes the deletion of contaminated and unnecessary files.

REQUIRES: DELETION_REPORT.json (generated by scan_deletion_targets.py)
"""

from pathlib import Path
import json
import os
import shutil

PROJECT_ROOT = Path(".")


def load_deletion_report():
    """Load deletion report."""
    report_file = PROJECT_ROOT / "DELETION_REPORT.json"

    if not report_file.exists():
        raise FileNotFoundError(
            "DELETION_REPORT.json not found. "
            "Run scripts/scan_deletion_targets.py first."
        )

    with open(report_file, 'r') as f:
        return json.load(f)


def execute_deletion(report, dry_run=False):
    """Execute deletion of files."""

    deleted_count = 0
    failed_count = 0
    deleted_size = 0

    print("=" * 80)
    if dry_run:
        print("DRY RUN - Files will NOT be deleted")
    else:
        print("EXECUTING DELETION")
    print("=" * 80)

    for file_info in report["files"]:
        filepath = Path(file_info["path"])
        category = file_info["category"]
        size = file_info["size_bytes"]

        if not filepath.exists():
            print(f"⚠️  SKIP: {filepath} (already deleted)")
            continue

        try:
            if dry_run:
                print(f"🔍 WOULD DELETE: {filepath} ({size:,} bytes) [{category}]")
                deleted_count += 1
                deleted_size += size
            else:
                # Actually delete
```

```
            if filepath.is_file():
                filepath.unlink()
            elif filepath.is_dir():
                shutil.rmtree(filepath)

            print(f"✓ DELETED: {filepath} ({size:,} bytes) [{category}]")
            deleted_count += 1
            deleted_size += size

        except Exception as e:
            print(f"✗ FAILED: {filepath} - {str(e)}")
            failed_count += 1

    # Summary
    print("\n" + "=" * 80)
    print("DELETION SUMMARY")
    print("=" * 80)
    print(f"{'Mode:':<20} {'DRY RUN' if dry_run else 'EXECUTED'}")
    print(f"{'Files deleted:':<20} {deleted_count}/{report['total_files']}")
    print(f"{'Failed:':<20} {failed_count}")
    print(f"{'Space freed:':<20} {deleted_size:,} bytes ({deleted_size / 1024 / 1024:.2f}
MB)")

    return deleted_count, failed_count


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

    # Load report
    report = load_deletion_report()

    print("Deletion plan loaded:")
    print(f"  Total files: {report['total_files']}")
    print(f"  Total size: {report['total_size_bytes']:,} bytes
({report['total_size_bytes'] / 1024 / 1024:.2f} MB)")
    print()

    # Ask for confirmation
    print("⚠  WARNING: This will permanently delete 44 files (4.67 MB)")
    print("   All files are backed up in MIGRATION_ARTIFACTS_FAKE_TO_REAL/")
    print()
    response = input("Type 'DELETE' to confirm deletion: ")

    if response != "DELETE":
        print("\n✗ Deletion cancelled.")
        print("To see what would be deleted, run with --dry-run:")
        print("  python3 scripts/execute_deletion.py --dry-run")
        return

    # Execute deletion
    print()
    deleted, failed = execute_deletion(report, dry_run=False)

    # Final status
    print()
    if failed == 0 and deleted == report['total_files']:
        print("✓ DELETION COMPLETE - All files successfully deleted")
        print()
        print("Next steps:")
        print("1. Commit the deletions: git add -A && git commit -m 'chore: Phase 1
massive deletion'")
        print("2. Proceed to Phase 2: Folder restructuring")
    else:
        print(f"⚠  DELETION INCOMPLETE - {failed} failures")


if __name__ == "__main__":
```

```
    import sys

    # Check for dry-run flag
    if "--dry-run" in sys.argv:
        report = load_deletion_report()
        execute_deletion(report, dry_run=True)
    else:
        main()

===== FILE: scripts/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("__", "_")

    return f"{namespace}.{method_key}"

# Generate contracts
output_dir = PROJECT_ROOT / "config" / "executor_contracts"
output_dir.mkdir(parents=True, exist_ok=True)

generated_count = 0
errors = []

for question in micro_questions:
    base_slot = question["base_slot"]
    question_id = question["question_id"]

    try:
        # Get methods for this question
        if base_slot not in methods_mapping:
            errors.append(f"{base_slot}: No methods found in mapping")
            continue

        question_methods = methods_mapping[base_slot]
        method_count = len(question_methods)

        # Create contract from template
        contract = json.loads(json.dumps(template))  # Deep copy
```

```
        # Update identity
        contract["identity"]["base_slot"] = base_slot
        contract["identity"]["question_id"] = question_id
        contract["identity"]["dimension_id"] = question.get("identity",
{}).get("dimension_id", "DIM01")
        contract["identity"]["policy_area_id"] = question.get("policy_area_id", "PA01")
        contract["identity"]["created_at"] =
datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%SZ")

        # Update executor_binding
        executor_class = base_slot.replace("-", "_") + "_Executor"
        contract["executor_binding"]["executor_class"] = executor_class

        # Build methods array
        methods_array = []
        for i, method_info in enumerate(question_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": get_provides_key(class_name, method_name),
                "role": get_method_role(method_name)
            })

        # Update method_binding
        contract["method_binding"] = {
            "orchestration_mode": "multi_method_pipeline",
            "method_count": method_count,
            "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": "≥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"]["met
adata"]["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/saaaaaa/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"✓ {generated_count:3d}/300 {base_slot:8s} ({method_count:2d} methods) →
{output_path.name}")

    except Exception as e:
        errors.append(f"{base_slot}: {str(e)}")
        print(f"✗ {base_slot}: Error - {str(e)}")

# Summary
print("\n" + "="*80)
print(f"✓ Generated: {generated_count}/300 contracts")
if errors:
    print(f"✗ Errors: {len(errors)}")
    for error in errors[:10]:  # Show first 10 errors
        print(f"  - {error}")
else:
    print("🎉 All contracts generated successfully!")
print("="*80)

# Save generation log
log_path = PROJECT_ROOT / "docs" / "contract_generation_log.json"
with open(log_path, 'w') as f:
    json.dump({
        "timestamp": datetime.utcnow().isoformat(),
        "generated_count": generated_count,
        "total_expected": 300,
        "errors": errors,
        "template_used": str(template_path),
        "output_directory": str(output_dir)
    }, f, indent=2)

print(f"\n📄 Generation log saved: {log_path}")

===== 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_d1q1_final.py =====
#!/usr/bin/env python3
"""
Generate the final D1-Q1.v3.CANONICAL.json with:
1. Correct method_binding.methods array (17 methods)
2. Complete human_answer_structure section
"""

import json
from pathlib import Path

PROJECT_ROOT = Path(__file__).parent.parent

# Load current contract
current_contract_path = PROJECT_ROOT / "config" / "executor_contracts" /
"D1-Q1.v3.CANONICAL.json"
with open(current_contract_path) as f:
    contract = 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)

d1q1_methods = methods_mapping["D1-Q1"]

# Define role mapping for each method
method_roles = {
    "diagnose_critical_links": "critical_link_diagnosis",
    "_analyze_link_text": "link_context_analysis",
    "process": "industrial_policy_pattern_processing",
    "_match_patterns_in_sentences": "sentence_level_pattern_matching",
```

```
    "_extract_point_evidence": "point_evidence_extraction",
    "_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_años",
                                "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_años": {"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": "según datos de DANE para el año 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_años",
            "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_años": 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 autonomía económica",
        "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_años": {
            "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 and 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 contains 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"✓ 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/generate_dependency_files.py =====
```
#!/usr/bin/env python3
"""
Generate comprehensive dependency files for SAAAAAA project.

Generates:
- requirements-core.txt: Core runtime dependencies with exact pins
- requirements-dev.txt: Development and testing dependencies
- requirements-optional.txt: Optional runtime dependencies
- requirements-constraints.txt: Full constraint file with all transitive deps
- Updated pyproject.toml with proper dependency groups
"""

import json
import sys
from pathlib import Path
from typing import Dict, List, Set


def load_audit_report(project_root: Path) -> Dict:
    """Load the dependency audit report."""
    report_file = project_root / "dependency_audit_report.json"
    if not report_file.exists():
        print("Error: dependency_audit_report.json not found. Run audit_dependencies.py
first.")
        sys.exit(1)

    with open(report_file, 'r') as f:
        return json.load(f)


def get_current_versions(requirements_file: Path) -> Dict[str, str]:
    """Extract package versions from existing requirements.txt."""
    versions = {}

    if not requirements_file.exists():
        return versions

    with open(requirements_file, 'r') as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith('#'):
                continue

            if '==' in line:
                pkg, ver = line.split('==', 1)
                versions[pkg.lower().strip()] = ver.strip()

    return versions


# Core runtime dependencies with known compatibility for Python 3.10-3.12
CORE_DEPENDENCIES = {
    # Web frameworks
    "flask": "3.0.3",
    "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")


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

    with open(output_file, 'w') as f:
        f.write("# Development & Testing Dependencies - Exact Pins\n")
        f.write("# Generated by scripts/generate_dependency_files.py\n")
        f.write("# Install with: pip install -r requirements-dev.txt\n\n")
        f.write("# Include core dependencies\n")
        f.write("-r requirements-core.txt\n\n")

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


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

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

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


def generate_all_requirements(output_file: Path, versions: Dict[str, str]):
    """Generate requirements-all.txt combining all dependencies."""
    print(f"Generating {output_file}...")

    with open(output_file, 'w') as f:
        f.write("# All Dependencies - For Complete Installation\n")
        f.write("# Generated by scripts/generate_dependency_files.py\n")
        f.write("# Install with: pip install -r requirements-all.txt\n\n")

        f.write("# Core Runtime\n")
        f.write("-r requirements-core.txt\n\n")

        f.write("# Optional Runtime\n")
        f.write("-r requirements-optional.txt\n\n")

        f.write("# Development\n")
        for pkg, version in sorted(DEV_DEPENDENCIES.items()):
            f.write(f"{pkg}=={version}\n")
        f.write("\n")
```

```
        f.write("# Documentation\n")
        for pkg, version in sorted(DOCS_DEPENDENCIES.items()):
            f.write(f"{pkg}=={version}\n")


def generate_constraints_file(output_file: Path):
    """Generate constraints.txt file."""
    print(f"Generating {output_file}...")

    with open(output_file, 'w') as f:
        f.write("# Constraints file - Exact version pins for ALL dependencies\n")
        f.write("# Generated by scripts/generate_dependency_files.py\n")
        f.write("# Use with: pip install -c constraints.txt -r requirements.txt\n\n")
        f.write("# This file prevents dependency conflicts by pinning all versions\n")
        f.write("# including transitive dependencies.\n\n")
        f.write("# To regenerate transitive dependencies:\n")
        f.write("# 1. Install all requirements in a clean venv\n")
        f.write("# 2. Run: pip freeze > constraints-full.txt\n")
        f.write("# 3. Review and merge into this file\n\n")

        # Combine all dependencies
        all_deps = {}
        all_deps.update(CORE_DEPENDENCIES)
        all_deps.update(OPTIONAL_DEPENDENCIES)
        all_deps.update(DEV_DEPENDENCIES)
        all_deps.update(DOCS_DEPENDENCIES)

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


def main():
    """Main entry point."""
    project_root = Path(__file__).parent.parent

    # Load existing versions from requirements.txt
    existing_versions = get_current_versions(project_root / "requirements.txt")

    # Generate all requirement files
    generate_core_requirements(project_root / "requirements-core.txt", existing_versions)
    generate_optional_requirements(project_root / "requirements-optional.txt",
existing_versions)
    generate_dev_requirements(project_root / "requirements-dev.txt", existing_versions)
    generate_docs_requirements(project_root / "requirements-docs.txt", existing_versions)
    generate_all_requirements(project_root / "requirements-all.txt", existing_versions)
    generate_constraints_file(project_root / "constraints-new.txt")

    print("\n" + "="*80)
    print("Dependency files generated successfully!")
    print("="*80)
    print("\nGenerated files:")
    print("  - requirements-core.txt: Core runtime dependencies")
    print("  - requirements-optional.txt: Optional runtime dependencies")
    print("  - requirements-dev.txt: Development dependencies (includes core)")
    print("  - requirements-docs.txt: Documentation dependencies")
    print("  - requirements-all.txt: All dependencies combined")
    print("  - constraints-new.txt: Version constraints for all packages")
    print("\nNext steps:")
    print("  1. Review the generated files")
    print("  2. Test installation: pip install -r requirements-core.txt")
    print("  3. Run verification: make deps:verify")
    print("  4. Replace old files if everything works")


if __name__ == "__main__":
    main()
```

```
===== FILE: scripts/generate_graphs.py =====
import matplotlib.pyplot as plt
import numpy as np
import argparse
import os

# AtroZ Dashboard Color Palette
colors = {
    'bg': '#0A0A0A',
    'ink': '#E5E7EB',
    'red': '#C41E3A',
    'blue': '#00D4FF',
    'green': '#39FF14',
    'copper': '#B2642E',
    'copper_oxide': '#17A589'
}

plt.rcParams['figure.facecolor'] = colors['bg']
plt.rcParams['axes.facecolor'] = colors['bg']
plt.rcParams['text.color'] = colors['ink']
plt.rcParams['axes.labelcolor'] = colors['ink']
plt.rcParams['xtick.color'] = colors['ink']
plt.rcParams['ytick.color'] = colors['ink']
plt.rcParams['axes.edgecolor'] = colors['copper']
plt.rcParams['font.family'] = 'JetBrains Mono'

def get_text(lang, key):
    """Gets the text for the given language and key."""
    text = {
        'en': {
            'control_flow_title': 'Control-Flow Graph',
            'input_config': 'Input config',
```