```
================================================================================
F.A.R.F.A.N PIPELINE CODE AUDIT - BATCH 30
================================================================================
Generated: 2025-12-07T06:17:37.702282
Files in this batch: 11
================================================================================


================================================================================
FILE: tools/scan_boundaries.py
================================================================================

#!/usr/bin/env python3
"""
AST Scanner for Core Module Boundary Violations

Scans Python modules for:
1. I/O operations (open, json.load/dump, pickle, pandas read_*, etc.)
2. __main__ blocks
3. Side effects on import
4. subprocess, requests, click usage

Usage:
    python tools/scan_boundaries.py --root src --fail-on=io,subprocess,requests,main
                                     --allow-path src/examples src/cli
                                     --sarif out/boundaries.sarif --json out/violations.json

Exit code 0 if clean, 1 if violations found.
"""

import argparse
import ast
import json
import sys
from datetime import datetime
from pathlib import Path

class BoundaryViolationVisitor(ast.NodeVisitor):
    """AST visitor to detect boundary violations in core modules."""

    # I/O function names to detect
    IO_FUNCTIONS = {
        ’open’, ’read’, ’write’,
        ’load’, ’dump’, ’loads’, ’dumps’,
        ’read_csv’, ’read_excel’, ’read_json’, ’read_sql’, ’read_parquet’,
        ’to_csv’, ’to_excel’, ’to_json’, ’to_sql’, ’to_parquet’,
        ’read_text’, ’write_text’, ’read_bytes’, ’write_bytes’,
    }

    # Module names that indicate I/O
    IO_MODULES = {
        ’pickle’, ’json’, ’yaml’, ’toml’, ’csv’,
    }

    # Subprocess/network modules
    SUBPROCESS_MODULES = {’subprocess’, ’os.system’}
    NETWORK_MODULES = {’requests’, ’urllib’, ’http’, ’httpx’}
    CLI_MODULES = {’click’, ’argparse’, ’sys.argv’}

    def __init__(self, filename: str) -> None:
        self.filename = filename
        self.violations: list[dict[str, any]] = []
        self.has_main_block = False
        self.io_calls: list[tuple[int, str]] = []
        self.subprocess_calls: list[tuple[int, str]] = []
        self.network_calls: list[tuple[int, str]] = []
        self.cli_usage: list[tuple[int, str]] = []

    def visit_If(self, node: ast.If) -> None:
        """Detect if __name__ == ’__main__’ blocks."""
        # Check for __name__ == ’__main__’ pattern
        if isinstance(node.test, ast.Compare) and isinstance(node.test.left, ast.Name) and node.test.left.id == ’__name__’:
            for comparator in node.test.comparators:
                if isinstance(comparator, ast.Constant) and comparator.value == ’__main__’:
                    self.has_main_block = True
                    self.violations.append({
                        ’type’: ’main_block’,
                        ’line’: node.lineno,
                        ’message’: f’__main__ block found at line {node.lineno}’
                    })
        self.generic_visit(node)

    def visit_Call(self, node: ast.Call) -> None:
        """Detect I/O function calls."""
        # Direct function calls
        if isinstance(node.func, ast.Name):
            if node.func.id in self.IO_FUNCTIONS:
                self.io_calls.append((node.lineno, node.func.id))
                self.violations.append({
                    ’type’: ’io_call’,
                    ’line’: node.lineno,
                    ’function’: node.func.id,
                    ’message’: f’I/O operation {node.func.id}() at line {node.lineno}’
                })

        # Module.function calls (e.g., json.load)
        elif isinstance(node.func, ast.Attribute) and isinstance(node.func.value, ast.Name):
            module_name = node.func.value.id
            func_name = node.func.attr

            # Check for I/O
            if module_name in self.IO_MODULES or func_name in self.IO_FUNCTIONS:
                self.io_calls.append((node.lineno, f’{module_name}.{func_name}’))
                self.violations.append({
                    ’type’: ’io_call’,
                    ’line’: node.lineno,
                    ’function’: f’{module_name}.{func_name}’,
                    ’message’: f’I/O operation {module_name}.{func_name}() at line {node.lineno}’
                })

            # Check for subprocess
            if module_name in self.SUBPROCESS_MODULES:
                self.subprocess_calls.append((node.lineno, f’{module_name}.{func_name}’))
                self.violations.append({
                    ’type’: ’subprocess_call’,
                    ’line’: node.lineno,
                    ’function’: f’{module_name}.{func_name}’,
                    ’message’: f’Subprocess call {module_name}.{func_name}() at line {node.lineno}’
                })

            # Check for network
            if module_name in self.NETWORK_MODULES:
                self.network_calls.append((node.lineno, f’{module_name}.{func_name}’))
                self.violations.append({
                    ’type’: ’network_call’,
                    ’line’: node.lineno,
                    ’function’: f’{module_name}.{func_name}’,
                    ’message’: f’Network call {module_name}.{func_name}() at line {node.lineno}’
                })

        self.generic_visit(node)

    def visit_With(self, node: ast.With) -> None:
        """Detect ’with open(...)’ patterns."""
        for item in node.items:
            if (isinstance(item.context_expr, ast.Call) and
                isinstance(item.context_expr.func, ast.Name) and
                item.context_expr.func.id == ’open’):
                self.io_calls.append((node.lineno, ’open (with)’))
                self.violations.append({
                    ’type’: ’io_call’,
                    ’line’: node.lineno,
                    ’function’: ’open’,
                    ’message’: f’I/O operation: with open(...) at line {node.lineno}’
                })
        self.generic_visit(node)

def scan_file(filepath: Path) -> dict[str, any]:
    """Scan a single Python file for boundary violations."""
    try:
        with open(filepath, encoding=’utf-8’) as f:
            source = f.read()
    except Exception as e:
        return {
            ’file’: str(filepath),
            ’error’: f’Could not read file: {e}’,
            ’violations’: [],
            ’clean’: False
        }

    try:
        tree = ast.parse(source, filename=str(filepath))
    except SyntaxError as e:
        return {
            ’file’: str(filepath),
            ’error’: f’Syntax error: {e}’,
            ’violations’: [],
            ’clean’: False
        }

    visitor = BoundaryViolationVisitor(str(filepath))
    visitor.visit(tree)

    return {
        ’file’: str(filepath),
        ’violations’: visitor.violations,
        ’has_main_block’: visitor.has_main_block,
        ’io_call_count’: len(visitor.io_calls),
        ’clean’: len(visitor.violations) == 0,
        ’error’: None
    }

def scan_directory(directory: Path, pattern: str = ’*.py’) -> list[dict[str, any]]:
    """Scan all Python files in a directory."""
    results = []
    for filepath in sorted(directory.rglob(pattern)):
        # Skip __pycache__ and test files
        if ’__pycache__’ in str(filepath) or ’test_’ in filepath.name:
            continue
        results.append(scan_file(filepath))
    return results

def generate_sarif_report(results: list[dict[str, any]], tool_version: str = "1.0.0") -> dict:
    """Generate SARIF 2.1.0 format report for GitHub annotations."""
    sarif = {
        "version": "2.1.0",
        "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/Schemata/sarif-schema-2.1.0.json",
        "runs": [
            {
                "tool": {
                    "driver": {
                        "name": "BoundaryScanner",
                        "version": tool_version,
                        "informationUri": "https://github.com/kkkkknhh/SAAAAAA",
                        "rules": [
                            {
                                "id": "IO_VIOLATION",
                                "name": "I/O Operation in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not perform I/O operations"
                                },
                                "fullDescription": {
                                    "text": "All I/O operations must be performed through ports and adapters"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "MAIN_BLOCK",
                                "name": "__main__ Block in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not contain __main__ blocks"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "SUBPROCESS_VIOLATION",
                                "name": "Subprocess Call in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not call subprocess"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            },
                            {
                                "id": "NETWORK_VIOLATION",
                                "name": "Network Call in Core Module",
                                "shortDescription": {
                                    "text": "Core modules must not make network calls"
                                },
                                "defaultConfiguration": {
                                    "level": "error"
                                }
                            }
                        ]
                    }
                },
                "results": []
            }
        ]
    }

    for result in results:
        if not result[’clean’] and not result.get(’error’):
            for violation in result[’violations’]:
                rule_id = {
                    ’io_call’: ’IO_VIOLATION’,
                    ’main_block’: ’MAIN_BLOCK’,
                    ’subprocess_call’: ’SUBPROCESS_VIOLATION’,
                    ’network_call’: ’NETWORK_VIOLATION’
                }.get(violation[’type’], ’IO_VIOLATION’)

                sarif_result = {
                    "ruleId": rule_id,
                    "level": "error",
                    "message": {
                        "text": violation[’message’]
                    },
                    "locations": [
                        {
                            "physicalLocation": {
                                "artifactLocation": {
                                    "uri": result[’file’],
                                    "uriBaseId": "%SRCROOT%"
                                },
                                "region": {
                                    "startLine": violation[’line’],
                                    "startColumn": 1
                                }
                            }
                        }
                    ]
                }
                sarif[’runs’][0][’results’].append(sarif_result)

    return sarif

def generate_json_report(results: list[dict[str, any]]) -> dict:
    """Generate JSON violations report keyed by file, line, and node type."""
    violations_by_file = {}

    for result in results:
        if not result[’clean’]:
            file_path = result[’file’]
            violations_by_file[file_path] = {
                ’violations’: result[’violations’],
                ’has_main_block’: result.get(’has_main_block’, False),
                ’io_call_count’: result.get(’io_call_count’, 0),
                ’error’: result.get(’error’)
            }

    return {
        ’timestamp’: datetime.now().isoformat(),
        ’total_files_scanned’: len(results),
        ’files_with_violations’: len(violations_by_file),
        ’total_violations’: sum(len(r[’violations’]) for r in results),
        ’violations_by_file’: violations_by_file
    }

def should_allow_path(filepath: Path, allowed_paths: list[str]) -> bool:
    """Check if filepath is in any of the allowed paths."""
    filepath_str = str(filepath)
    return any(allowed in filepath_str for allowed in allowed_paths)

def print_report(results: list[dict[str, any]], fail_on_types: set[str] | None = None) -> int:
    """Print scan results and return exit code."""
    total_files = len(results)
    clean_files = sum(1 for r in results if r[’clean’])
    total_violations = sum(len(r[’violations’]) for r in results)

    if fail_on_types is None:
        fail_on_types = {’io_call’, ’main_block’, ’subprocess_call’, ’network_call’}

    print("=" * 80)
    print("CORE MODULE BOUNDARY SCAN REPORT")
    print("=" * 80)
    print(f"\nFiles scanned: {total_files}")
    print(f"Clean files: {clean_files}")
    print(f"Files with violations: {total_files - clean_files}")
    print(f"Total violations: {total_violations}")
    print()

    if total_violations == 0:
        print("â\234\205 All files are clean! No boundary violations detected.")
        return 0

    print("â\235\214 Violations found:\n")

    # Count violations by type
    violation_counts = {}
    for result in results:
        if not result[’clean’]:
            for violation in result[’violations’]:
                vtype = violation[’type’]
                violation_counts[vtype] = violation_counts.get(vtype, 0) + 1

    print("Violation summary:")
    for vtype, count in sorted(violation_counts.items()):
        marker = "â\235\214" if vtype in fail_on_types else "â\232 ï¸\217 "
        print(f"  {marker} {vtype}: {count}")
    print()

    for result in results:
        if not result[’clean’]:
            print(f"\n{result[’file’]}")
            if result.get(’error’):
                print(f"  ERROR: {result[’error’]}")
            else:
                for violation in result[’violations’]:
                    marker = "â\235\214" if violation[’type’] in fail_on_types else "â\232 ï¸\217 "
                    print(f"  {marker} Line {violation[’line’]}: {violation[’message’]}")

    print("\n" + "=" * 80)
    print("REMEDIATION:")
    print("- Move all __main__ blocks to examples/ directory")
    print("- Move all I/O operations to orchestrator/factory.py")
    print("- Core modules should be pure libraries receiving data via contracts")
    print("=" * 80)

    # Determine if we should fail based on fail_on_types
    should_fail = any(
        violation[’type’] in fail_on_types
        for result in results
        for violation in result[’violations’]
    )

    return 1 if should_fail else 0

def main() -> int:
    """Main entry point."""
    parser = argparse.ArgumentParser(
        description=’Scan Python modules for boundary violations’,
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog="""
Examples:
  # Basic scan
  python tools/scan_boundaries.py --root src/farfan_pipeline/core

# Fail on specific violations only
python tools/scan_boundaries.py --root src --fail-on io,main

# Allow specific paths
python tools/scan_boundaries.py --root src --fail-on io,subprocess,requests,main \\
                                 --allow-path src/examples src/cli

# Generate SARIF and JSON reports
python tools/scan_boundaries.py --root src/farfan_pipeline/core \\
                                 --sarif out/boundaries.sarif \\
                                 --json out/violations.json
      """
  )

  parser.add_argument(
      ’--root’,
      type=str,
      required=True,
      help=’Root directory to scan’
  )

  parser.add_argument(
      ’--fail-on’,
      type=str,
      default=’io,main,subprocess,network’,
      help=’Comma-separated list of violation types to fail on (io, main, subprocess, network)’
  )

  parser.add_argument(
      ’--allow-path’,
      nargs=’+’,
      default=[],
      help=’Paths to exclude from scanning (e.g., src/examples src/cli)’
  )

  parser.add_argument(
      ’--sarif’,
      type=str,
      help=’Output SARIF report to this file’
  )

  parser.add_argument(
      ’--json’,
      type=str,
      help=’Output JSON violations report to this file’
  )

  # Legacy positional argument support
  if len(sys.argv) == 2 and not sys.argv[1].startswith(’--’):
      # Old style: python scan_boundaries.py <directory>
      target_path = Path(sys.argv[1])
      fail_on_types = {’io_call’, ’main_block’, ’subprocess_call’, ’network_call’}
      allowed_paths = []
      sarif_output = None
      json_output = None
  else:
        args = parser.parse_args()
        target_path = Path(args.root)
        fail_on_types = set()
        for vtype in args.fail_on.split(’,’):
            vtype = vtype.strip()
            if vtype == ’io’:
                fail_on_types.add(’io_call’)
            elif vtype == ’main’:
                fail_on_types.add(’main_block’)
            elif vtype == ’subprocess’:
                fail_on_types.add(’subprocess_call’)
            elif vtype == ’network’:
                fail_on_types.add(’network_call’)

        allowed_paths = args.allow_path
        sarif_output = args.sarif
        json_output = args.json

    if not target_path.exists():
        print(f"Error: Directory {target_path} does not exist")
        return 1

    results = scan_directory(target_path)

    # Filter results based on allowed paths
    if allowed_paths:
        filtered_results = []
        for result in results:
            if not should_allow_path(Path(result[’file’]), allowed_paths):
                filtered_results.append(result)
            else:
                # Mark as clean if in allowed path
                result[’clean’] = True
                result[’violations’] = []
                filtered_results.append(result)
        results = filtered_results

    # Generate SARIF report if requested
    if sarif_output:
        sarif_path = Path(sarif_output)
        sarif_path.parent.mkdir(parents=True, exist_ok=True)
        sarif_data = generate_sarif_report(results)
        with open(sarif_path, ’w’, encoding=’utf-8’) as f:
            json.dump(sarif_data, f, indent=2)
        print(f"SARIF report written to {sarif_output}")

    # Generate JSON report if requested
    if json_output:
        json_path = Path(json_output)
        json_path.parent.mkdir(parents=True, exist_ok=True)
        json_data = generate_json_report(results)
        with open(json_path, ’w’, encoding=’utf-8’) as f:
            json.dump(json_data, f, indent=2)
        print(f"JSON report written to {json_output}")

    return print_report(results, fail_on_types)
```

```

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



================================================================================
FILE: tools/scan_core_purity.py
================================================================================

#!/usr/bin/env python3
"""
Core Purity Scanner - Ensures core modules follow functional purity principles.

Checks:
1. No I/O operations in core modules (print, open, file operations)
2. No __main__ blocks in core modules
3. No direct database or network calls
"""

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

try:
    from farfan_pipeline.config.paths import PROJECT_ROOT
except Exception:  # pragma: no cover - bootstrap fallback
    PROJECT_ROOT = Path(__file__).resolve().parents[1]

# Directories that must maintain purity
CORE_PATHS = [
    "src/farfan_pipeline/core",
]

# Forbidden operations (allowing open for config loading, but not print/input)
FORBIDDEN_FUNCTIONS = {
    "print", "input",
}

FORBIDDEN_IMPORTS = {
    "requests", "urllib", "socket", "sqlalchemy",
}


class PurityChecker(ast.NodeVisitor):
    """AST visitor to detect impure operations."""

    def __init__(self, filepath: Path):
        self.filepath = filepath
        self.violations: List[Tuple[int, str]] = []

    def visit_Call(self, node: ast.Call):
        """Check for forbidden function calls."""
        if isinstance(node.func, ast.Name):
            if node.func.id in FORBIDDEN_FUNCTIONS:
                self.violations.append(
                    (node.lineno, f"Forbidden function: {node.func.id}")
                )
        self.generic_visit(node)

    def visit_If(self, node: ast.If):
        """Check for __main__ blocks."""
        if isinstance(node.test, ast.Compare):
            if isinstance(node.test.left, ast.Name):
                if node.test.left.id == "__name__":
                    self.violations.append(
                        (node.lineno, "Forbidden __main__ block in core module")
                    )
        self.generic_visit(node)

    def visit_Import(self, node: ast.Import):
        """Check for forbidden imports."""
        for alias in node.names:
            if any(forbidden in alias.name for forbidden in FORBIDDEN_IMPORTS):
                self.violations.append(
                    (node.lineno, f"Forbidden import: {alias.name}")
                )
        self.generic_visit(node)

    def visit_ImportFrom(self, node: ast.ImportFrom):
        """Check for forbidden imports."""
        if node.module:
            if any(forbidden in node.module for forbidden in FORBIDDEN_IMPORTS):
                self.violations.append(
                    (node.lineno, f"Forbidden import from: {node.module}")
                )
        self.generic_visit(node)


def check_file_purity(filepath: Path) -> List[Tuple[int, str]]:
    """Check a single file for purity violations."""
    try:
        with open(filepath, "r", encoding="utf-8") as f:
            tree = ast.parse(f.read(), filename=str(filepath))

        checker = PurityChecker(filepath)
        checker.visit(tree)
        return checker.violations
    except SyntaxError as e:
        return [(e.lineno or 0, f"Syntax error: {e.msg}")]
    except Exception as e:
        return [(0, f"Error parsing file: {e}")]


def main() -> int:
    """Scan all core modules for purity violations."""
    repo_root = PROJECT_ROOT
    violations_found = False

    for core_path_str in CORE_PATHS:
        core_path = repo_root / core_path_str

        if not core_path.exists():
            print(f"â\232 ï¸\217  Core path not found: {core_path}")
            continue

        print(f"Scanning {core_path_str}...")

        for py_file in core_path.rglob("*.py"):
            violations = check_file_purity(py_file)

            if violations:
                violations_found = True
                rel_path = py_file.relative_to(repo_root)
                print(f"\nâ\235\214 {rel_path}")
                for lineno, msg in violations:
                    print(f"  Line {lineno}: {msg}")

    if violations_found:
        print("\nâ\235\214 Core purity violations detected")
        return 1
    else:
        print("â\234\223 All core modules are pure")
        return 0


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



================================================================================
FILE: tools/signal_ecosystem_validator.py
================================================================================

#!/usr/bin/env python3
"""
Signal Ecosystem Validator - AnÃ¡lisis basado en AST

Este script analiza SOLO imports reales en tu cÃ³digo para determinar
quÃ© archivos de signals estÃ¡n realmente en uso.

NO hace suposiciones. NO adivina. Solo lee el AST.
"""

import ast
from pathlib import Path
from typing import Dict, Set, List
from dataclasses import dataclass, field


@dataclass
class ModuleAnalysis:
    """Resultado del anÃ¡lisis de un mÃ³dulo."""

    filepath: Path
    imports_from_signals: Set[str] = field(default_factory=set)
    imported_by: Set[str] = field(default_factory=set)
    is_dead_code: bool = False
    reason: str = ""


class SignalEcosystemValidator:
    """Validador del ecosistema de signals basado en AST."""

    SIGNAL_MODULES = [
        ’signal_aliasing’,
        ’signal_cache_invalidation’,
        ’signal_consumption’,
        ’signal_context_scoper’,
        ’signal_contract_validator’,
        ’signal_evidence_extractor’,
        ’signal_fallback_fusion’,
        ’signal_intelligence_layer’,
        ’signal_loader’,
        ’signal_quality_metrics’,
        ’signal_registry’,
        ’signal_resolution’,
        ’signal_semantic_expander’,
        ’signals’,
    ]

    def __init__(self, src_root: Path):
        """Inicializa validador con ruta raÃ-z del cÃ³digo."""
        self.src_root = src_root
        self.orchestrator_dir = src_root / "core" / "orchestrator"
        self.results: Dict[str, ModuleAnalysis] = {}

    def analyze_file(self, filepath: Path) -> ModuleAnalysis:
        """Analiza un archivo Python y extrae sus imports reales."""
        analysis = ModuleAnalysis(filepath=filepath)

        try:
            with open(filepath, ’r’, encoding=’utf-8’) as f:
                tree = ast.parse(f.read(), filename=str(filepath))

            # Extraer imports
            for node in ast.walk(tree):
                if isinstance(node, ast.ImportFrom):
                    module = node.module or ’’

                    # Â¿Importa desde signals?
                    for signal_mod in self.SIGNAL_MODULES:
                        if signal_mod in module:
                            analysis.imports_from_signals.add(signal_mod)

                elif isinstance(node, ast.Import):
                    for alias in node.names:
                        name = alias.name
                        for signal_mod in self.SIGNAL_MODULES:
                            if signal_mod in name:
                                analysis.imports_from_signals.add(signal_mod)

        except Exception as e:
            analysis.reason = f"Error parsing: {e}"

        return analysis

    def scan_ecosystem(self) -> Dict[str, ModuleAnalysis]:
        """Escanea todo el ecosistema de orchestrator."""

        # 1. Analizar TODOS los archivos de signals primero
        print("ð\237\223\202 Scanning signal modules...")
        for signal_mod in self.SIGNAL_MODULES:
            signal_file = self.orchestrator_dir / f"{signal_mod}.py"
            if signal_file.exists():
                self.results[signal_mod] = self.analyze_file(signal_file)
                print(f"  â\234\223 {signal_mod}.py")

        # 2. Construir grafo de dependencias ENTRE signals
        print("\nð\237\224\227 Building signal-to-signal dependency graph...")
        for signal_mod, analysis in self.results.items():
            for imported_signal in analysis.imports_from_signals:
                if imported_signal in self.results:
                    self.results[imported_signal].imported_by.add(f"{signal_mod}.py")
                    print(f"  {signal_mod} â\206\222 imports â\206\222 {imported_signal}")

        # 3. Analizar archivos principales del orchestrator
        print("\nð\237\223\213 Scanning main orchestrator files...")
        main_files = [
            ’evidence_assembler.py’, ’evidence_registry.py’, ’evidence_validator.py’, ’executor_config.py’,  ’executor_profiler.py’, ’executors.py’, ’core.py’,
            ’chunk_router.py’,
            ’resource_integration.py’, ’resource_manager.py’, ’task_planner.py’, ’executor_config.py’,
            ’factory.py’,
            ’base_executor_with_contract.py’,
            ’irrigation_synchronizer.py’,
            ’arg_router.py’,
            ’method_registry.py’,
        ]

        for main_file in main_files:
            main_path = self.orchestrator_dir / main_file
            if main_path.exists():
                analysis = self.analyze_file(main_path)

                # Marcar quÃ© signals son importados por archivos principales
                for signal_mod in analysis.imports_from_signals:
                    if signal_mod in self.results:
                        self.results[signal_mod].imported_by.add(main_file)
                        print(f"  {main_file} â\206\222 imports â\206\222 {signal_mod}")

        # 4. Identificar ROOT nodes (importados por cÃ³digo principal)
        print("\nð\237\214³ Identifying root nodes...")
        root_nodes = set()
        for signal_mod, analysis in self.results.items():
            for importer in analysis.imported_by:
                if not importer.startswith(’signal_’):
                    root_nodes.add(signal_mod)
                    print(f"  ROOT: {signal_mod} (used by {importer})")

        # 5. Propagar vitalidad desde roots usando BFS
        print("\nâ\231»ï¸\217  Propagating liveness from roots...")
        alive = set(root_nodes)
        queue = list(root_nodes)

        while queue:
            current = queue.pop(0)
            if current not in self.results:
                continue

            # Marcar dependencias transitivas como vivas
            for dep in self.results[current].imports_from_signals:
                if dep not in alive and dep in self.results:
                    alive.add(dep)
                    queue.append(dep)
                    print(f"  {dep} is alive (transitively via {current})")

        # 6. Marcar cÃ³digo muerto (no alcanzable desde roots)
        print("\nð\237\222\200 Marking dead code...")
        for signal_mod, analysis in self.results.items():
            if signal_mod not in alive:
                analysis.is_dead_code = True
                analysis.reason = "Not reachable from any main orchestrator file"
                print(f"  DEAD: {signal_mod}")

        return self.results

    def generate_report(self) -> str:
        """Genera reporte de anÃ¡lisis."""

        lines = []
        lines.append("=" * 80)
        lines.append("SIGNAL ECOSYSTEM ANALYSIS - COMPLETE GRAPH TRAVERSAL")
        lines.append("=" * 80)
        lines.append("")

        # ROOT NODES: Importados directamente por cÃ³digo principal
        root_nodes = {}
        for mod, analysis in self.results.items():
            for importer in analysis.imported_by:
                if not importer.startswith(’signal_’):
                    root_nodes[mod] = analysis
                    break

        if root_nodes:
            lines.append("ð\237\214³ ROOT NODES (imported by main orchestrator code)")
            lines.append("-" * 80)
            for mod, analysis in sorted(root_nodes.items()):
                importers = [i for i in analysis.imported_by if not i.startswith(’signal_’)]
                lines.append(f"  {mod}.py")
                lines.append(f"    â\206\222 Used by: {’, ’.join(sorted(importers))}")
            lines.append("")

        # TRANSITIVE: Importados indirectamente (signal â\206\222 signal)
        transitive = {}
        for mod, analysis in self.results.items():
            if not analysis.is_dead_code and mod not in root_nodes:
                transitive[mod] = analysis

        if transitive:
            lines.append("ð\237\224\227 TRANSITIVE DEPENDENCIES (signal â\206\222 signal chain)")
            lines.append("-" * 80)
            for mod, analysis in sorted(transitive.items()):
                signal_importers = [i for i in analysis.imported_by if i.startswith(’signal_’)]
                lines.append(f"  {mod}.py")
                lines.append(f"    â\206\222 Used by signals: {’, ’.join(sorted(signal_importers))}")
            lines.append("")

        # DEAD CODE: No alcanzable desde roots
        dead = {k: v for k, v in self.results.items() if v.is_dead_code}
        if dead:
            lines.append("ð\237\222\200 DEAD CODE (not reachable from any main file)")
            lines.append("-" * 80)
            for mod in sorted(dead.keys()):
                lines.append(f"  {mod}.py")
            lines.append("")

        # EstadÃ-sticas
        lines.append("ð\237\223\212 STATISTICS")
        lines.append("-" * 80)
        lines.append(f"Total signal modules analyzed: {len(self.results)}")
        lines.append(f"Root nodes (directly used): {len(root_nodes)}")
        lines.append(f"Transitive dependencies: {len(transitive)}")
        lines.append(f"Dead code modules: {len(dead)}")
        lines.append(f"Dead code percentage: {100 * len(dead) / len(self.results):.1f}%")
        lines.append("")

        # Recomendaciones
        lines.append("â\234\205 RECOMMENDATIONS")
        lines.append("-" * 80)
        lines.append("1. KEEP ALL: Root nodes + transitive dependencies")
        lines.append("2. REVIEW: Dead code modules - verify with git blame/history")
        lines.append("3. MOVE: Experimental/proposed code to /experiments or /proposals")
        lines.append("4. DELETE: Confirmed abandoned code after team review")
        lines.append("")

        # Dependency chains
        lines.append("ð\237\223\210 LONGEST DEPENDENCY CHAINS")
        lines.append("-" * 80)
        chains = self._find_longest_chains()
        for i, chain in enumerate(chains[:5], 1):
            lines.append(f"{i}. {’ â\206\222 ’.join(chain)}")
        lines.append("")

        return "\n".join(lines)

    def _find_longest_chains(self) -> List[List[str]]:
        """Encuentra las cadenas de dependencias mÃ¡s largas."""
        chains = []

        # Empezar desde cada root node
        for mod, analysis in self.results.items():
            has_non_signal_importer = any(
                not i.startswith(’signal_’) for i in analysis.imported_by
            )
            if has_non_signal_importer:
                chains.extend(self._dfs_chains(mod, []))

        # Ordenar por longitud descendente
        chains.sort(key=len, reverse=True)
        return chains

    def _dfs_chains(self, node: str, current_chain: List[str]) -> List[List[str]]:
        """DFS para encontrar todas las cadenas desde un nodo."""
        if node in current_chain:  # Evitar ciclos
            return [current_chain + [node]]

        new_chain = current_chain + [node]

        if node not in self.results:
            return [new_chain]

        deps = self.results[node].imports_from_signals
        if not deps:
            return [new_chain]

        all_chains = []
        for dep in deps:
            all_chains.extend(self._dfs_chains(dep, new_chain))

        return all_chains if all_chains else [new_chain]

    def get_dependency_graph(self) -> str:
        """Genera grafo de dependencias en formato texto."""

        lines = []
        lines.append("DEPENDENCY GRAPH")
        lines.append("-" * 80)

        for mod, analysis in sorted(self.results.items()):
            if not analysis.is_dead_code:
                lines.append(f"\n{mod}")
                if analysis.imports_from_signals:
                    for dep in sorted(analysis.imports_from_signals):
                        lines.append(f"  â\224\224â\224\200â\224\200 depends on: {dep}")
                if analysis.imported_by:
                    for user in sorted(analysis.imported_by):
                        lines.append(f"  â\206\221 used by: {user}")

        return "\n".join(lines)


def main():
    """Ejecuta anÃ¡lisis del ecosistema."""

    # AJUSTA ESTA RUTA a tu proyecto
    src_root = Path("src/farfan_pipeline")

    if not src_root.exists():
        print(f"â\235\214 ERROR: {src_root} no existe")
        print("Ajusta la ruta ’src_root’ en main()")
        return

    validator = SignalEcosystemValidator(src_root)

    print("Escaneando ecosystem...")
    validator.scan_ecosystem()

    print(validator.generate_report())
    print(validator.get_dependency_graph())

    # Exportar resultados
    report_file = Path("signal_ecosystem_analysis.txt")
    with open(report_file, ’w’, encoding=’utf-8’) as f:
        f.write(validator.generate_report())
        f.write("\n\n")
        f.write(validator.get_dependency_graph())

    print(f"\nâ\234\205 Reporte guardado en: {report_file}")


if __name__ == "__main__":
    main()



================================================================================
FILE: tools/testing/__init__.py
================================================================================

"""Testing utilities."""



================================================================================
FILE: tools/testing/boot_check.py
================================================================================

#!/usr/bin/env python3
"""
Boot check script for CI and pre-production environments.

Validates that all modules load correctly, runtime validators initialize,
and the registry is complete without ClassNotFoundError.

Usage:
    python tools/testing/boot_check.py
    python tools/testing/boot_check.py --verbose
"""

import importlib
import sys
import traceback
from pathlib import Path

# Add project root to Python path

# Modules to validate
CORE_MODULES = [
    "orchestrator",
    "scoring",
    "recommendation_engine",
    "validation_engine",
    "policy_processor",
    "embedding_policy",
    "semantic_chunking_policy",
]

OPTIONAL_MODULES = [
    "derek_beach",
    "contradiction_deteccion",
    "teoria_cambio",
    "financiero_viabilidad_tablas",
    "macro_prompts",
    "micro_prompts",
]

def check_module_import(module_name: str, verbose: bool = False) -> tuple[bool, str]:
    """
    Try to import a module and return success status.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print(f"  Importing {module_name}...", end=" ")

        importlib.import_module(module_name)

        if verbose:
            print("â\234\223")

        return True, ""
    except ModuleNotFoundError as e:
        error = f"Module not found: {e}"
        if verbose:
            print(f"â\234\227 {error}")
        return False, error
    except ImportError as e:
        error = f"Import error: {e}"
        if verbose:
            print(f"â\234\227 {error}")
        return False, error
    except Exception as e:
        error = f"Unexpected error: {e}"
        if verbose:
            print(f"â\234\227 {error}")
        if verbose:
            traceback.print_exc()
        return False, error

def check_registry_validation(verbose: bool = False) -> tuple[bool, str]:
    """
    Validate that the orchestrator registry loads without ClassNotFoundError.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print("  Validating orchestrator registry...", end=" ")

        # Try to import and access the registry
        from farfan_pipeline.core.orchestrator import registry

        # Try to validate all classes (if method exists)
        if hasattr(registry, ’validate_all_classes’):
            registry.validate_all_classes()

        if verbose:
            print("â\234\223")

        return True, ""
    except NameError as e:
        if "ClassNotFoundError" in str(e) or "not defined" in str(e):
            error = f"ClassNotFoundError in registry: {e}"
            if verbose:
                print(f"â\234\227 {error}")
            return False, error
        raise
    except AttributeError as e:
        # Module or registry doesn’t exist - return as informational
        if verbose:
            print(f"â\232  Registry validation not available: {e}")
        return True, ""  # Don’t fail if registry validation not implemented
    except Exception as e:
        error = f"Registry validation error: {e}"
        if verbose:
            print(f"â\234\227 {error}")
            traceback.print_exc()
        return False, error

def check_runtime_validators(verbose: bool = False) -> tuple[bool, str]:
    """
    Validate that runtime validators initialize correctly.

    Returns:
        Tuple of (success, error_message)
    """
    try:
        if verbose:
            print("  Initializing runtime validators...", end=" ")

        # Try to import and initialize validators
        from farfan_pipeline.validation.validation_engine import RuntimeValidator

        validator = RuntimeValidator()

        # Try to run health check if available
        if hasattr(validator, ’health_check’):
            validator.health_check()

        if verbose:
            print("â\234\223")

        return True, ""
    except ImportError:
        # validation_engine doesn’t exist or RuntimeValidator not available
        if verbose:
            print("â\232  Runtime validators not available (skipping)")
        return True, ""  # Don’t fail if not implemented
    except Exception as e:
        error = f"Runtime validator initialization error: {e}"
        if verbose:
            print(f"â\234\227 {error}")
            traceback.print_exc()
        return False, error

def run_boot_checks(verbose: bool = False) -> int:
    """
    Run all boot checks.

    Returns:
        Exit code (0 = success, 1 = failure)
    """
    print("=" * 60)
    print("Boot Check - Module and Runtime Validation")
    print("=" * 60)

    all_passed = True
    failed_checks = []

    # Check core modules
    print("\nChecking core modules:")
    core_failed = []
    for module in CORE_MODULES:
        success, error = check_module_import(module, verbose)
        if not success:
            all_passed = False
            core_failed.append(f"{module}: {error}")
            failed_checks.append(f"Core module {module} failed to load")

    if not verbose:
        if core_failed:
            print(f"  â\234\227 {len(core_failed)} core module(s) failed to load")
            for failure in core_failed[:3]:
                print(f"    - {failure}")
            if len(core_failed) > 3:
                print(f"    ... and {len(core_failed) - 3} more")
        else:
            print(f"  â\234\223 All {len(CORE_MODULES)} core modules loaded successfully")

    # Check optional modules
    print("\nChecking optional modules:")
    optional_failed = []
    for module in OPTIONAL_MODULES:
        success, error = check_module_import(module, verbose)
        if not success:
            optional_failed.append(f"{module}: {error}")
            # Don’t fail overall for optional modules

    if not verbose:
        loaded_count = len(OPTIONAL_MODULES) - len(optional_failed)
        print(f"  â\234\223 {loaded_count}/{len(OPTIONAL_MODULES)} optional modules loaded")
        if optional_failed:
            print(f"  â\232  {len(optional_failed)} optional module(s) not available")

    # Check registry
    print("\nChecking orchestrator registry:")
    success, error = check_registry_validation(verbose)
    if not success:
        all_passed = False
        failed_checks.append(f"Registry validation failed: {error}")
    elif not verbose:
        print("  â\234\223 Registry validation passed")

    # Check runtime validators
    print("\nChecking runtime validators:")
    success, error = check_runtime_validators(verbose)
    if not success:
        all_passed = False
        failed_checks.append(f"Runtime validator initialization failed: {error}")
    elif not verbose:
        print("  â\234\223 Runtime validators initialized successfully")

    # Summary
    print("\n" + "=" * 60)
    if all_passed:
        print("â\234\223 All boot checks PASSED")
        print("=" * 60)
        return 0
    else:
        print("â\234\227 Some boot checks FAILED")
        print("\nFailed checks:")
        for check in failed_checks:
            print(f"  - {check}")
        print("=" * 60)
        return 1

def main() -> None:
    verbose = "--verbose" in sys.argv or "-v" in sys.argv

    exit_code = run_boot_checks(verbose)
    sys.exit(exit_code)

```

```
if __name__ == "__main__":
    main()



================================================================================
FILE: tools/validation/__init__.py
================================================================================

"""Validation tools."""



================================================================================
FILE: tools/validation/validate_build_hygiene.py
================================================================================

#!/usr/bin/env python3
"""
Validation script for build hygiene checklist.
Verifies that the repository follows all build hygiene requirements.
"""

import re
import sys
from pathlib import Path

def check_python_version_pin() -> bool:
    """Check that Python version is pinned to 3.12.x."""
    print("â\234\223 Checking Python version pin...")

    # Check .python-version
    python_version_file = Path(".python-version")
    if not python_version_file.exists():
        print("  â\234\227 Missing .python-version file")
        return False

    version = python_version_file.read_text().strip()
    if not version.startswith("3.12"):
        print(f"  â\234\227 .python-version should be 3.12.x, got {version}")
        return False

    # Check pyproject.toml
    pyproject = Path("pyproject.toml").read_text()
    if ’requires-python = "˜=3.12.0"’ not in pyproject:
        print("  â\234\227 pyproject.toml should have requires-python = \"˜=3.12.0\"")
        return False

    if ’pythonVersion = "3.12"’ not in pyproject:
        print("  â\234\227 pyproject.toml should have pythonVersion = \"3.12\"")
        return False

    print("  â\234\223 Python version properly pinned to 3.12.x")
    return True

def check_pinned_dependencies() -> bool:
    """Check that all dependencies are pinned to exact versions."""
    print("â\234\223 Checking dependency pinning...")

    requirements = Path("requirements.txt")
    if not requirements.exists():
        print("  â\234\227 Missing requirements.txt")
        return False

    constraints = Path("constraints.txt")
    if not constraints.exists():
        print("  â\234\227 Missing constraints.txt")
        return False

    # Check for wildcards or open ranges in requirements.txt
    content = requirements.read_text()
    lines = [line.strip() for line in content.split('\n')
             if line.strip() and not line.strip().startswith('#')]

    bad_patterns = []
    for line in lines:
        # Check for wildcard or open ranges in version specifiers
        # Look for these patterns after package name and optional extras
        # Match patterns like: package>=1.0, package˜=1.0, package>1.0, package<2.0, package*
        if (re.search(r'(>=|˜=|>|<|\*)', line) and
            re.search(r'[^\[]*(\[.*\])?\s*(>=|˜=|>|<|\*)', line)):
            # Further validate it's in version spec position, not in package name
            # Package name format: name[extras]==version
            bad_patterns.append(line)

    if bad_patterns:
        print("  â\234\227 Found wildcards or open ranges in requirements.txt:")
        for pattern in bad_patterns:
            print(f"    - {pattern}")
        return False

    print(f"  â\234\223 All {len(lines)} dependencies properly pinned with exact versions")
    print("  â\234\223 constraints.txt exists")
    return True

def check_directory_structure() -> bool:
    """Check that required directories exist with __init__.py files."""
    print("â\234\223 Checking directory structure...")

    required_dirs = [
        "orchestrator",
        "executors",
        "contracts",
        "tests",
        "tools",
        "examples",
        "src/farfan_pipeline/core",
    ]

    missing_dirs = []
    missing_init = []

    for dir_path in required_dirs:
        path = Path(dir_path)
        if not path.exists():
            missing_dirs.append(dir_path)
        elif not (path / "__init__.py").exists():
            missing_init.append(dir_path)

    if missing_dirs:
        print("  â\234\227 Missing directories:")
        for d in missing_dirs:
            print(f"    - {d}")
        return False

    if missing_init:
        print("  â\234\227 Missing __init__.py in:")
        for d in missing_init:
            print(f"    - {d}")
        return False

    print("  â\234\223 All required directories exist with __init__.py files")
    return True

def check_pythonpath_config() -> bool:
    """Check that setup.py exists for proper PYTHONPATH configuration."""
    print("â\234\223 Checking PYTHONPATH configuration...")

    setup_py = Path("setup.py")
    if not setup_py.exists():
        print("  â\234\227 Missing setup.py for pip install -e .")
        return False

    content = setup_py.read_text()
    if ’find_packages’ not in content:
        print("  â\234\227 setup.py should use find_packages()")
        return False

    print("  â\234\223 setup.py exists for editable installation")
    return True

def check_centralized_config() -> bool:
    """Check that centralized configuration exists."""
    print("â\234\223 Checking centralized configuration...")

    settings = Path("orchestrator/settings.py")
    if not settings.exists():
        print("  â\234\227 Missing orchestrator/settings.py")
        return False

    env_example = Path(".env.example")
    if not env_example.exists():
        print("  â\234\227 Missing .env.example")
        return False

    gitignore = Path(".gitignore")
    if gitignore.exists():
        content = gitignore.read_text()
        if ".env" not in content:
            print("  â\234\227 .gitignore should exclude .env files")
            return False

    print("  â\234\223 Centralized config exists (orchestrator/settings.py, .env.example)")
    print("  â\234\223 .env properly excluded in .gitignore")
    return True

def main() -> int:
    """Run all validation checks."""
    print("=" * 60)
    print("Build Hygiene Validation")
    print("=" * 60)
    print()

    checks = [
        check_python_version_pin,
        check_pinned_dependencies,
        check_directory_structure,
        check_pythonpath_config,
        check_centralized_config,
    ]

    results = []
    for check in checks:
        try:
            results.append(check())
            print()
        except Exception as e:
            print(f"  â\234\227 Error running check: {e}")
            results.append(False)
            print()

    print("=" * 60)
    if all(results):
        print("â\234\223 All build hygiene checks passed!")
        print("=" * 60)
        return 0
    else:
        failed = sum(1 for r in results if not r)
        print(f"â\234\227 {failed} check(s) failed")
        print("=" * 60)
        return 1

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



================================================================================
FILE: tools/validation/validate_scoring_parity.py
================================================================================

#!/usr/bin/env python3
"""
Validate scoring parity across modalities.
```

```

This script ensures that:
1. Normalization formulas are correct for each modality
2. Quality thresholds are identical across all modalities
3. Boundary conditions produce correct quality levels
4. No modality has an unfair advantage at quality boundaries

Usage:
    python tools/validation/validate_scoring_parity.py
    python tools/validation/validate_scoring_parity.py --verbose
"""

import sys

# Quality thresholds (must be identical across all modalities)
QUALITY_THRESHOLDS = {
    "EXCELENTE": 0.85,
    "BUENO": 0.70,
    "ACEPTABLE": 0.55,
    "INSUFICIENTE": 0.00
}

# Modality score ranges
MODALITY_RANGES = {
    "TYPE_A": (0, 4),
    "TYPE_B": (0, 3),
    "TYPE_C": (0, 3),
    "TYPE_D": (0, 3),
    "TYPE_E": (0, 3),
    "TYPE_F": (0, 3),
}

def normalize_score(raw_score: float, modality: str) -> float:
    """Normalize a raw score to [0, 1] range based on modality."""
    min_score, max_score = MODALITY_RANGES[modality]
    if raw_score < min_score or raw_score > max_score:
        raise ValueError(f"Score {raw_score} out of range for {modality}: [{min_score}, {max_score}]")
    return (raw_score - min_score) / (max_score - min_score)

def determine_quality_level(normalized_score: float) -> str:
    """Determine quality level from normalized score."""
    # Use small epsilon for floating point comparison
    epsilon = 1e-9
    if normalized_score >= QUALITY_THRESHOLDS["EXCELENTE"] - epsilon:
        return "EXCELENTE"
    elif normalized_score >= QUALITY_THRESHOLDS["BUENO"] - epsilon:
        return "BUENO"
    elif normalized_score >= QUALITY_THRESHOLDS["ACEPTABLE"] - epsilon:
        return "ACEPTABLE"
    else:
        return "INSUFICIENTE"

def test_normalization_formulas() -> bool:
    """Test that normalization formulas are correct."""
    print("Testing normalization formulas...")

    test_cases = [
        ("TYPE_A", 0, 0.0),
        ("TYPE_A", 2, 0.5),
        ("TYPE_A", 4, 1.0),
        ("TYPE_B", 0, 0.0),
        ("TYPE_B", 1.5, 0.5),
        ("TYPE_B", 3, 1.0),
        ("TYPE_C", 0, 0.0),
        ("TYPE_C", 1.5, 0.5),
        ("TYPE_C", 3, 1.0),
    ]

    passed = 0
    failed = 0

    for modality, raw, expected in test_cases:
        actual = normalize_score(raw, modality)
        if abs(actual - expected) < 0.001:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  â\234\223 {modality}: {raw} â\206\222 {actual:.3f} (expected {expected:.3f})")
        else:
            failed += 1
            print(f"  â\234\227 {modality}: {raw} â\206\222 {actual:.3f} (expected {expected:.3f})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_parity_at_thresholds() -> bool:
    """Test that all modalities produce the same quality level at threshold scores."""
    print("\nTesting parity at quality thresholds...")

    # Calculate equivalent raw scores for each threshold
    test_cases = []
    for quality, threshold in QUALITY_THRESHOLDS.items():
        if quality == "INSUFICIENTE":
            continue  # Skip lower bound

        for modality, (min_score, max_score) in MODALITY_RANGES.items():
            raw_score = min_score + threshold * (max_score - min_score)
            test_cases.append((modality, raw_score, quality))

    passed = 0
    failed = 0

    for modality, raw_score, expected_quality in test_cases:
        normalized = normalize_score(raw_score, modality)
        actual_quality = determine_quality_level(normalized)

        if actual_quality == expected_quality:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  â\234\223 {modality} at {raw_score:.2f} â\206\222 {actual_quality}")
        else:
            failed += 1
            print(f"  â\234\227 {modality} at {raw_score:.2f} â\206\222 {actual_quality} (expected {expected_quality})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_boundary_conditions() -> bool:
    """Test boundary conditions (just above/below thresholds)."""
    print("\nTesting boundary conditions...")

    # Test scores just below and just above EXCELENTE threshold

    test_cases = [
        # Just below EXCELENTE (should be BUENO)
        ("TYPE_A", 3.396, "BUENO"),  # 3.396/4 = 0.849
        ("TYPE_B", 2.547, "BUENO"),  # 2.547/3 = 0.849

        # Just at EXCELENTE threshold
        ("TYPE_A", 3.4, "EXCELENTE"),  # 3.4/4 = 0.85
        ("TYPE_B", 2.55, "EXCELENTE"),  # 2.55/3 = 0.85

        # Just above EXCELENTE
        ("TYPE_A", 3.404, "EXCELENTE"),  # 3.404/4 = 0.851
        ("TYPE_B", 2.553, "EXCELENTE"),  # 2.553/3 = 0.851
    ]

    passed = 0
    failed = 0

    for modality, raw_score, expected_quality in test_cases:
        normalized = normalize_score(raw_score, modality)
        actual_quality = determine_quality_level(normalized)

        if actual_quality == expected_quality:
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  â\234\223 {modality}: {raw_score:.3f} (norm={normalized:.4f}) â\206\222 {actual_quality}")
        else:
            failed += 1
            print(f"  â\234\227 {modality}: {raw_score:.3f} (norm={normalized:.4f}) â\206\222 {actual_quality} (expected {expected_quality})")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def test_no_unfair_advantage() -> bool:
    """Test that no modality has an unfair advantage at boundaries."""
    print("\nTesting for unfair advantages...")

    # For each quality threshold, calculate the "difficulty" (raw score needed)
    # relative to the maximum possible score
    difficulties = {}

    for quality, threshold in QUALITY_THRESHOLDS.items():
        if quality == "INSUFICIENTE":
            continue

        difficulties[quality] = {}
        for modality, (min_score, max_score) in MODALITY_RANGES.items():
            raw_needed = min_score + threshold * (max_score - min_score)
            relative_difficulty = raw_needed / max_score
            difficulties[quality][modality] = relative_difficulty

    passed = 0
    failed = 0

    for quality, modality_difficulties in difficulties.items():
        # All modalities should have the same relative difficulty
        values = list(modality_difficulties.values())
        max_diff = max(values) - min(values)

        if max_diff < 0.001:  # Allow 0.1% variance
            passed += 1
            if "--verbose" in sys.argv:
                print(f"  â\234\223 {quality}: all modalities have equal difficulty (max diff: {max_diff:.6f})")
        else:
            failed += 1
            print(f"  â\234\227 {quality}: modalities have unequal difficulty (max diff: {max_diff:.6f})")
            for modality, diff in modality_difficulties.items():
                print(f"      {modality}: {diff:.6f}")

    print(f"  Passed: {passed}/{passed + failed}")
    return failed == 0

def main() -> int:
    """Run all parity validation tests."""
    print("=" * 60)
    print("Scoring Parity Validation")
    print("=" * 60)

    all_passed = True

    # Run all tests
    all_passed &= test_normalization_formulas()
    all_passed &= test_parity_at_thresholds()
    all_passed &= test_boundary_conditions()
    all_passed &= test_no_unfair_advantage()

    print("\n" + "=" * 60)
    if all_passed:
        print("â\234\223 All parity validation tests PASSED")
        print("=" * 60)
        return 0
    else:
        print("â\234\227 Some parity validation tests FAILED")
        print("=" * 60)
        return 1

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



================================================================================
FILE: update_executors_memory.py
```

```
================================================================================

#!/usr/bin/env python3
"""
Script to systematically add memory safety metrics to all executor return statements.
"""
import re

def update_executors_file(filepath: str) -> None:
    with open(filepath, ’r’, encoding=’utf-8’) as f:
        content = f.read()

    pattern = r’(\s+)"execution_metrics": \{\s+("methods_count":.*?\n\s+"all_succeeded":.*?)\s+\}’

    def replacer(match):
        indent = match.group(1)
        metrics_content = match.group(2)

        if "memory_safety" in metrics_content:
            return match.group(0)

        return (
            f’{indent}"execution_metrics": {{\n’
            f’{indent}    {metrics_content},\n’
            f’{indent}    "memory_safety": self._get_memory_metrics_summary()\n’
            f’{indent}}}’
        )

    updated_content = re.sub(pattern, replacer, content)

    with open(filepath, ’w’, encoding=’utf-8’) as f:
        f.write(updated_content)

    print(f"Updated {filepath}")

if __name__ == "__main__":
    update_executors_file("src/farfan_pipeline/core/orchestrator/executors.py")



================================================================================
FILE: verify_canonical_inventory.py
================================================================================

#!/usr/bin/env python3
"""Verify the structure of generated canonical inventory files."""

import json
import sys


def verify_canonical_inventory():
    print("Verifying canonical_method_inventory.json...")
    with open("canonical_method_inventory.json") as f:
        data = json.load(f)

    assert "methods" in data, "Missing ’methods’ key"
    assert "metadata" in data, "Missing ’metadata’ key"

    metadata = data["metadata"]
    assert "total_methods" in metadata
    assert "scan_timestamp" in metadata
    assert "source_directory" in metadata

    methods = data["methods"]
    assert len(methods) > 0, "No methods found"

    sample_method = next(iter(methods.values()))
    required_keys = [
        "canonical_name",
        "file_path",
        "line_number",
        "class_name",
        "role",
        "is_executor",
        "signature",
    ]
    for key in required_keys:
        assert key in sample_method, f"Missing key ’{key}’ in method"

    assert (
        "parameters" in sample_method["signature"]
    ), "Missing ’parameters’ in signature"

    print(f"  â\234\223 Valid structure with {len(methods)} methods")
    return True


def verify_statistics():
    print("Verifying method_statistics.json...")
    with open("method_statistics.json") as f:
        data = json.load(f)

    required_keys = [
        "total_methods",
        "total_executors",
        "by_role",
        "by_module",
        "executor_distribution",
    ]
    for key in required_keys:
        assert key in data, f"Missing key ’{key}’"

    assert data["total_methods"] > 0
    assert data["total_executors"] >= 0
    assert len(data["by_role"]) > 0
    assert len(data["by_module"]) > 0

    print("  â\234\223 Valid structure")
    print(f"    Total methods: {data[’total_methods’]}")
    print(f"    Total executors: {data[’total_executors’]}")
    print(f"    Roles: {list(data[’by_role’].keys())}")
    return True


def verify_excluded():
    print("Verifying excluded_methods.json...")
    with open("excluded_methods.json") as f:
        data = json.load(f)

    assert "excluded_methods" in data, "Missing ’excluded_methods’ key"
    assert "exclusion_reason" in data, "Missing ’exclusion_reason’ key"
    assert "total_excluded" in data, "Missing ’total_excluded’ key"

    assert data["exclusion_reason"] == "never calibrate"
    assert data["total_excluded"] == len(data["excluded_methods"])

    if len(data["excluded_methods"]) > 0:
        sample = data["excluded_methods"][0]
        required_keys = [
            "canonical_id",
            "reason",
            "method_name",
            "file_path",
            "line_number",
        ]
        for key in required_keys:
            assert key in sample, f"Missing key ’{key}’ in excluded method"

    print(f"  â\234\223 Valid structure with {data[’total_excluded’]} excluded methods")
    return True


def main():
    try:
        verify_canonical_inventory()
        verify_statistics()
        verify_excluded()
        print("\nâ\234\205 All files verified successfully!")
        return 0
    except Exception as e:
        print(f"\nâ\235\214 Verification failed: {e}", file=sys.stderr)
        return 1


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



================================================================================
FILE: verify_inventory.py
================================================================================

#!/usr/bin/env python3
"""Standalone inventory verification script - does not require pytest"""

import json
import sys
from pathlib import Path
from typing import Any


def load_inventory() -> dict[str, Any] | None:
    """Load the inventory JSON file"""
    inventory_path = Path("methods_inventory_raw.json")

    if not inventory_path.exists():
        print(f"ERROR: Inventory file not found: {inventory_path}", file=sys.stderr)
        return None

    with open(inventory_path, encoding="utf-8") as f:
        return json.load(f)


MINIMUM_METHOD_COUNT = 200


def test_minimum_method_count(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify at least 200 methods in inventory"""
    total = inventory["metadata"]["total_methods"]
    if total < MINIMUM_METHOD_COUNT:
        return False, f"Insufficient methods: {total} < {MINIMUM_METHOD_COUNT}"
    return True, f"â\234\223 Method count: {total} >= {MINIMUM_METHOD_COUNT}"


def test_critical_files_present(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify methods from critical files are present"""
    methods = inventory["methods"]
    source_files = {m["source_file"] for m in methods}

    critical_files = [
        "derek_beach.py",
        "aggregation.py",
        "executors.py",
        "executors_contract.py",
    ]

    errors = []
    for critical_file in critical_files:
        found = any(critical_file in sf for sf in source_files)
        if not found:
            errors.append(f"Critical file not found: {critical_file}")

    if errors:
        return False, "\n  ".join(errors)
    return True, f"â\234\223 All {len(critical_files)} critical files present"


def test_critical_method_patterns(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify critical method patterns are present"""
    methods = inventory["methods"]
    canonical_ids = {m["canonical_identifier"] for m in methods}

    patterns_to_check = [
        "derek_beach",
        "aggregation",
        "executors",
    ]

    errors = []
    for pattern in patterns_to_check:
        found = any(pattern in cid.lower() for cid in canonical_ids)
        if not found:
            errors.append(f"No methods found matching pattern: {pattern}")

    if errors:
        return False, "\n  ".join(errors)
    return True, f"â\234\223 All {len(patterns_to_check)} critical patterns found"


def test_all_roles_present(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify all expected roles are present"""
    stats = inventory["statistics"]["by_role"]

    expected_roles = [
        "ingest",
        "processor",
        "analyzer",
        "extractor",
        "score",
        "utility",
        "orchestrator",
        "core",
        "executor",
    ]

    errors = []
    for role in expected_roles:
        if role not in stats:
            errors.append(f"Role not found: {role}")

    if errors:
        return False, "\n  ".join(errors)
    return True, f"â\234\223 All {len(expected_roles)} roles present"


def test_calibration_flags_set(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify calibration flags are properly set"""
    methods = inventory["methods"]

    calibration_count = sum(1 for m in methods if m["requiere_calibracion"])
    parametrization_count = sum(1 for m in methods if m["requiere_parametrizacion"])

    if calibration_count == 0:
        return False, "No methods flagged for calibration"
    if parametrization_count == 0:
        return False, "No methods flagged for parametrization"

    return (
        True,
        f"â\234\223 Calibration: {calibration_count}, Parametrization: {parametrization_count}",
    )


MIN_CANONICAL_ID_PARTS = 2
MAX_FORMAT_ERRORS = 5


def test_canonical_identifier_format(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify canonical identifiers follow module.Class.method format"""
    methods = inventory["methods"]
    errors = []

    for method in methods[:100]:  # Sample first 100
        cid = method["canonical_identifier"]
        parts = cid.split(".")

        if len(parts) < MIN_CANONICAL_ID_PARTS:
            errors.append(f"Invalid canonical ID format: {cid}")
            if len(errors) >= MAX_FORMAT_ERRORS:
                break

    if errors:
        return False, "\n  ".join(errors)
    return True, "â\234\223 All canonical identifiers properly formatted"


MIN_TAG_RATIO = 0.3


def test_epistemology_tags_present(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify epistemology tags are assigned"""
    methods = inventory["methods"]

    tagged_count = sum(1 for m in methods if m["epistemology_tags"])
    total = len(methods)

    if tagged_count == 0:
        return False, "No methods have epistemology tags"

    tag_ratio = tagged_count / total
    if tag_ratio < MIN_TAG_RATIO:
        return False, f"Too few methods tagged: {tag_ratio:.2%}"

    return True, f"â\234\223 Epistemology tags: {tagged_count}/{total} ({tag_ratio:.2%})"


MIN_DEREK_BEACH_METHODS = 10


def test_derek_beach_methods_complete(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify derek_beach.py methods are complete"""
    methods = inventory["methods"]
    derek_methods = [m for m in methods if "derek_beach" in m["source_file"]]

    if len(derek_methods) < MIN_DEREK_BEACH_METHODS:
        return False, f"Too few derek_beach methods: {len(derek_methods)}"

    required_patterns = ["_format_message", "to_dict", "_load_config", "classify_test"]
    errors = []

    for pattern in required_patterns:
        found = any(pattern in m["method_name"] for m in derek_methods)
        if not found:
            errors.append(f"Required derek_beach method not found: {pattern}")

    if errors:
        return False, "\n  ".join(errors)
    return (
        True,
        f"â\234\223 derek_beach.py: {len(derek_methods)} methods, all required patterns found",
    )


def test_aggregation_classes_present(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify aggregation classes are present"""
    methods = inventory["methods"]
    aggregation_methods = [
        m for m in methods if "aggregation" in m["source_file"].lower()
    ]

    if len(aggregation_methods) == 0:
        return False, "No aggregation methods found"

    required_classes = [
        "AreaPolicyAggregator",
        "ClusterAggregator",
        "DimensionAggregator",
    ]

    found_classes = {m["class_name"] for m in aggregation_methods if m["class_name"]}
    errors = []

    for req_class in required_classes:
        if req_class not in found_classes:
            errors.append(f"Required aggregation class not found: {req_class}")

    if errors:
        return False, "\n  ".join(errors)
    return (
        True,
        f"â\234\223 aggregation.py: {len(aggregation_methods)} methods, all classes found",
    )


MIN_EXECUTOR_METHODS = 5


def test_executor_methods_present(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify executor methods are present"""
    methods = inventory["methods"]
    executor_methods = [m for m in methods if "executor" in m["source_file"].lower()]

    if len(executor_methods) < MIN_EXECUTOR_METHODS:
        return False, f"Too few executor methods: {len(executor_methods)}"

    return True, f"â\234\223 executor files: {len(executor_methods)} methods"


def test_no_duplicate_canonical_ids(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify no duplicate canonical identifiers"""
    methods = inventory["methods"]
    canonical_ids = [m["canonical_identifier"] for m in methods]

    duplicates = [cid for cid in canonical_ids if canonical_ids.count(cid) > 1]

    if duplicates:
        unique_dupes = list(set(duplicates))[:5]
        return False, f"Duplicate canonical IDs found: {unique_dupes}"

    return True, "â\234\223 No duplicate canonical identifiers"


def test_layer_requirements_complete(inventory: dict[str, Any]) -> tuple[bool, str]:
    """Verify LAYER_REQUIREMENTS table is complete"""
    layer_requirements = inventory["layer_requirements"]

    expected_layers = [
        "ingest",
        "processor",
        "analyzer",
        "extractor",
        "score",
        "utility",
        "orchestrator",
        "core",
        "executor",
    ]

    errors = []
    for layer in expected_layers:
        if layer not in layer_requirements:
            errors.append(f"Layer missing from requirements: {layer}")
        elif "description" not in layer_requirements[layer]:
            errors.append(f"Layer missing description: {layer}")
        elif "typical_patterns" not in layer_requirements[layer]:
            errors.append(f"Layer missing typical_patterns: {layer}")

    if errors:
        return False, "\n  ".join(errors)
    return True, f"â\234\223 LAYER_REQUIREMENTS complete: {len(expected_layers)} layers"


def main() -> None:
    print("=" * 70)
    print("INVENTORY COMPLETENESS VERIFICATION")
    print("=" * 70)
    print()

    inventory = load_inventory()
    if inventory is None:
        sys.exit(1)

    tests = [
        ("Minimum method count", test_minimum_method_count),
        ("Critical files present", test_critical_files_present),
        ("Critical method patterns", test_critical_method_patterns),
        ("All roles present", test_all_roles_present),
        ("Calibration flags set", test_calibration_flags_set),
        ("Canonical identifier format", test_canonical_identifier_format),
        ("Epistemology tags present", test_epistemology_tags_present),
        ("derek_beach methods complete", test_derek_beach_methods_complete),
        ("aggregation classes present", test_aggregation_classes_present),
        ("executor methods present", test_executor_methods_present),
        ("No duplicate canonical IDs", test_no_duplicate_canonical_ids),
        ("LAYER_REQUIREMENTS complete", test_layer_requirements_complete),
    ]

    passed = 0
    failed = 0

    for test_name, test_func in tests:
        try:
            success, message = test_func(inventory)
            if success:
                print(f"PASS: {test_name}")
                print(f"      {message}")
                passed += 1
            else:
                print(f"FAIL: {test_name}")
                print(f"      {message}")
                failed += 1
        except Exception as e:
            print(f"ERROR: {test_name}")
            print(f"       {e}")
            failed += 1
        print()

    print("=" * 70)
    print(f"RESULTS: {passed} passed, {failed} failed")
    print("=" * 70)

    if failed > 0:
        sys.exit(1)
    else:
        print("\nâ\234\223â\234\223â\234\223 ALL TESTS PASSED â\234\223â\234\223â\234\223\n")
        sys.exit(0)


if __name__ == "__main__":
    main()
```