src/farfan_pipeline/phases/Phase_zero/coverage_gate.py

```
#!/usr/bin/env python3
"""
Coverage Enforcement Gate
=========================
Enforces hard-fail at <555 methods threshold + audit.json emission

Requirements:
- Count all public methods across Producer classes
- Generate audit.json with method counts and validation results
- Hard-fail if total methods < 555
- Include schema validation results
"""

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


def count_methods_in_class(filepath: Path, class_name: str) -> tuple[list[str], dict[str, int]]:
    """Count public and private methods in a class and return method names"""
    if not filepath.exists():
        return [], {"public": 0, "private": 0, "total": 0}

    with open(filepath, encoding='utf-8') as f:
        tree = ast.parse(f.read())

    method_names = []
    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):
                    method_names.append(item.name)
                    if not item.name.startswith('_'):
                        method_counts["public"] += 1
                    else:
                        method_counts["private"] += 1
                    method_counts["total"] += 1

    return method_names, method_counts

def validate_schema_exists(module_dir: Path) -> tuple[bool, list[str]]:
    """Validate that schema files exist for a module"""
    if not module_dir.exists():
        return False, []

    schema_files = list(module_dir.glob("*.schema.json"))
    return len(schema_files) > 0, [f.name for f in schema_files]

def count_file_methods(filepath: Path) -> tuple[int, int]:
    """Count all public and total methods in a file"""
    if not filepath.exists():
        return 0, 0

    with open(filepath, encoding='utf-8') as f:
        try:
            tree = ast.parse(f.read())
            public_methods = 0
            all_methods = 0

            for node in ast.walk(tree):
                if isinstance(node, ast.FunctionDef):
                    all_methods += 1
                    if not node.name.startswith('_'):
                        public_methods += 1

            return public_methods, all_methods
        except Exception as e:
            print(f"Error parsing {filepath}: {e}")
            return 0, 0

def count_all_methods() -> dict[str, any]:
    """Count all methods across all modules and producers"""

    # All files to analyze
    files_to_analyze = [
        "financiero_viabilidad_tablas.py",
        "Analyzer_one.py",
        "contradiction_deteccion.py",
        "embedding_policy.py",
        "teoria_cambio.py",
        "derek_beach.py",
        "policy_processor.py",
        "report_assembly.py",
        "semantic_chunking_policy.py"
    ]

    # Producer classes to check
    producers = {
        "SemanticChunkingProducer": "semantic_chunking_policy.py",
        "EmbeddingPolicyProducer": "embedding_policy.py",
        "DerekBeachProducer": "derek_beach.py",
        "ReportAssemblyProducer": "report_assembly.py"
    }

    results = {
        "timestamp": datetime.now().isoformat(),
        "files": {},
        "producers": {},
        "totals": {
            "file_public_methods": 0,
            "file_total_methods": 0,
            "producer_methods": 0,
            "threshold": 555,
            "meets_threshold": False
        },
        "schema_validation": {},
        "audit_status": "PENDING"
    }

    # Count file methods
    print("=" * 80)
    print("FILE METHOD COUNTS")
    print("=" * 80)

    for filepath_str in files_to_analyze:
        filepath = Path(filepath_str)
        public_methods, total_methods = count_file_methods(filepath)
        results["files"][filepath_str] = {
            "public_methods": public_methods,
            "total_methods": total_methods
        }
        results["totals"]["file_public_methods"] += public_methods
        results["totals"]["file_total_methods"] += total_methods
        print(f"{filepath_str:45} | {public_methods:4} public | {total_methods:4}
total")

    # Count Producer methods
    print("\n" + "=" * 80)
    print("PRODUCER METHOD COUNTS")
    print("=" * 80)

    for class_name, filepath in producers.items():
        methods, counts = count_methods_in_class(Path(filepath), class_name)
        results["producers"][class_name] = {
            "file": filepath,
            "methods": methods,
            "counts": counts,
            "public_methods": counts["public"]
        }
        results["totals"]["producer_methods"] += counts["public"]
        print(f"{class_name:45} | {counts['public']:3} public | {counts['private']:3}
private | {counts['total']:3} total")

    # Update meets_threshold
    results["totals"]["meets_threshold"] = (
        results["totals"]["file_total_methods"] >= 555
    )

    # Validate schemas
    print("\n" + "=" * 80)
    print("SCHEMA VALIDATION")
    print("=" * 80)

    schema_modules = [
        "semantic_chunking_policy",
        "embedding_policy",
        "derek_beach",
        "report_assembly"
    ]

    for module in schema_modules:
        module_dir = Path("schemas") / module
        has_schemas, schema_files = validate_schema_exists(module_dir)
        results["schema_validation"][module] = {
            "has_schemas": has_schemas,
            "schema_files": schema_files,
            "schema_count": len(schema_files)
        }
        status = "?" if has_schemas else "?"
        print(f"{module:35} | {status} | {len(schema_files)} schemas")

    # Determine audit status
    all_have_schemas = all(
        v["has_schemas"] for v in results["schema_validation"].values()
    )

    if results["totals"]["meets_threshold"] and all_have_schemas:
        results["audit_status"] = "PASS"
    else:
        results["audit_status"] = "FAIL"

    return results

def main() -> int:
    """Main entry point"""
    print("\n" + "=" * 80)
    print("COVERAGE ENFORCEMENT GATE")
    print("=" * 80 + "\n")

    # Count all methods
    results = count_all_methods()

    # Print summary
    print("\n" + "=" * 80)
    print("SUMMARY")
    print("=" * 80)
    print(f"Total file methods:      {results['totals']['file_total_methods']:4}")
    print(f"Total public methods:    {results['totals']['file_public_methods']:4}")
    print(f"Producer methods:        {results['totals']['producer_methods']:4}")
    print(f"Threshold:               {results['totals']['threshold']:4}")
    print(f"Meets threshold:         {results['totals']['meets_threshold']}")
    print(f"All schemas present:     {all(v['has_schemas'] for v in results['schema_validation'].values())}")
    print(f"Audit status:            {results['audit_status']}")
    # Save audit.json
    audit_path = Path("audit.json")
    with open(audit_path, 'w', encoding='utf-8') as f:
        json.dump(results, f, indent=2)

    print(f"\n? Audit results saved to {audit_path}")

    # Enforce hard-fail
    if not results['totals']['meets_threshold']:
        print("\n" + "=" * 80)
        print("? COVERAGE GATE FAILED")
        print("=" * 80)
        print(f"Required: {results['totals']['threshold']} methods")
        print(f"Found:    {results['totals']['file_total_methods']} methods")
                        print(f"Gap:                  {results['totals']['threshold']}   -
results['totals']['file_total_methods']} methods")
        print("=" * 80 + "\n")
        return 1

    # Check schema validation
    if not all(v['has_schemas'] for v in results['schema_validation'].values()):
        print("\n" + "=" * 80)
        print("? SCHEMA VALIDATION FAILED")
        print("=" * 80)
        for module, validation in results['schema_validation'].items():
            if not validation['has_schemas']:
                print(f"Missing schemas for: {module}")
        print("=" * 80 + "\n")
        return 1

    print("\n" + "=" * 80)
    print("? COVERAGE GATE PASSED")
    print("=" * 80)
    print(f"All {results['totals']['file_total_methods']} methods accounted for")
    print(f"{results['totals']['file_public_methods']} public methods available")
    print(f"{results['totals']['producer_methods']} producer methods exposed")
    print("All schema contracts validated")
    print("=" * 80 + "\n")

    return 0

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

src/farfan_pipeline/phases/Phase_zero/determinism.py

```
"""
Determinism Module - Consolidated Seed Management
==================================================

Provides centralized determinism enforcement for the F.A.R.F.A.N pipeline:
- Seed derivation from policy_unit_id and correlation_id
- RNG seeding for Python, NumPy, and advanced components
- Validation of seed application

This module consolidates:
- determinism_helpers.py (seed derivation, context manager)
- seed_factory.py (seed generation)
- Integration with global SeedRegistry

Author: Phase 0 Compliance Team
Version: 2.0.0
Specification: P00-EN v2.0 Section 3.4
"""

from __future__ import annotations

import hashlib
import hmac
import json
import os
import random
from contextlib import contextmanager
from dataclasses import dataclass
from typing import TYPE_CHECKING, Any, Iterator

if TYPE_CHECKING:
    from orchestration.seed_registry import SeedRegistry

try:
    import numpy as np
    NUMPY_AVAILABLE = True
except ImportError:
    NUMPY_AVAILABLE = False
    np = None  # type: ignore


# Required seeds for Phase 0 compliance
MANDATORY_SEEDS = ["python", "numpy"]
OPTIONAL_SEEDS = ["quantum", "neuromorphic", "meta_learner"]
ALL_SEEDS = MANDATORY_SEEDS + OPTIONAL_SEEDS


@dataclass(frozen=True)
class Seeds:
    """Container for seeds used in deterministic execution."""

    python: int
    numpy: int
    quantum: int | None = None
    neuromorphic: int | None = None
    meta_learner: int | None = None

    def to_dict(self) -> dict[str, int | None]:
        """Convert to dictionary for logging."""
        return {
            "python": self.python,
            "numpy": self.numpy,
            "quantum": self.quantum,
            "neuromorphic": self.neuromorphic,
            "meta_learner": self.meta_learner,
        }


def derive_seed_from_string(base_material: str, salt: bytes | None = None) -> int:
    """
    Derive deterministic seed from string using HMAC-SHA256.

    Args:
        base_material: String to hash (e.g., "PU_123:corr-1:python")
        salt: Optional salt for HMAC (default: fixed deployment salt)

    Returns:
        32-bit unsigned integer seed

    Example:
        >>> seed1 = derive_seed_from_string("PU_123:corr-1:python")
        >>> seed2 = derive_seed_from_string("PU_123:corr-1:python")
        >>> assert seed1 == seed2  # Deterministic
    """
    default_salt = b"FARFAN_PHASE0_DETERMINISTIC_SEED_2025"
    actual_salt = default_salt if salt is None else salt

    seed_hmac = hmac.new(
        key=actual_salt,
        msg=base_material.encode('utf-8'),
        digestmod=hashlib.sha256
    )

    seed_bytes = seed_hmac.digest()[:4]
    return int.from_bytes(seed_bytes, byteorder='big')


def derive_seed_from_parts(*parts: Any, salt: bytes | None = None) -> int:
    """
    Derive seed from arbitrary components via JSON serialization.

    Args:
        *parts: Components to hash (will be JSON-serialized)
        salt: Optional HMAC salt

    Returns:
        32-bit integer seed

    Example:
        >>> s1 = derive_seed_from_parts("PU_123", "corr-1", "python")
        >>> s2 = derive_seed_from_parts("PU_123", "corr-1", "python")
        >>> assert s1 == s2  # Deterministic
    """
    canonical = json.dumps(parts, sort_keys=True, separators=(",", ":"), ensure_ascii=False)
    return derive_seed_from_string(canonical, salt)


def apply_seeds_to_rngs(seeds: dict[str, int]) -> dict[str, bool]:
    """
    Apply seeds to all available RNGs.

    Args:
        seeds: Dictionary mapping component names to seed values

    Returns:
        Dictionary mapping component names to success status

    Raises:
        ValueError: If mandatory seeds are missing

    Example:
        >>> seeds = {"python": 12345, "numpy": 67890}
        >>> status = apply_seeds_to_rngs(seeds)
        >>> assert status["python"]
        >>> assert status["numpy"]
    """
    status = {}

    # Validate mandatory seeds
    missing = [s for s in MANDATORY_SEEDS if seeds.get(s) is None]
    if missing:
        raise ValueError(f"Missing mandatory seeds: {missing}")

    # Apply python seed (MANDATORY)
    python_seed = seeds["python"]
    random.seed(python_seed)
    status["python"] = True

    # Apply numpy seed (MANDATORY)
    if NUMPY_AVAILABLE and np is not None:
        numpy_seed = seeds["numpy"]
        np.random.seed(numpy_seed)
        status["numpy"] = True
    else:
        status["numpy"] = False

    # Apply optional seeds (best-effort)
    for component in OPTIONAL_SEEDS:
        seed = seeds.get(component)
        if seed is not None:
            # These components don't have global RNGs to seed yet
            # But we record them for future use
            status[component] = True
        else:
            status[component] = False

    return status


def validate_seed_application(seeds: dict[str, int], status: dict[str, bool]) ->
tuple[bool, list[str]]:
    """
    Validate that all required seeds were applied successfully.

    Args:
        seeds: Dictionary of seeds that were attempted
        status: Dictionary of application results from apply_seeds_to_rngs()

    Returns:
        Tuple of (success, errors)
        - success: True if all mandatory seeds applied
        - errors: List of error messages

    Example:
        >>> seeds = {"python": 12345, "numpy": 67890}
        >>> status = apply_seeds_to_rngs(seeds)
        >>> success, errors = validate_seed_application(seeds, status)
        >>> assert success
        >>> assert len(errors) == 0
    """
    errors = []

    # Check mandatory seeds
    for component in MANDATORY_SEEDS:
        if not status.get(component, False):
            errors.append(f"Failed to apply {component} seed")

    # Warn about optional seeds (but don't fail)
    missing_optional = [c for c in OPTIONAL_SEEDS if not status.get(c, False)]
    if missing_optional:
        # This is informational, not an error
        pass

    return len(errors) == 0, errors


def initialize_determinism_from_registry(
    seed_registry: SeedRegistry,
    policy_unit_id: str,
    correlation_id: str
) -> tuple[dict[str, int], dict[str, bool], list[str]]:
    """
    Initialize determinism using SeedRegistry (Phase 0.3 implementation).
```

```
This is the PRIMARY method for Phase 0 determinism initialization.

Args:
    seed_registry: Global seed registry instance
    policy_unit_id: Policy unit identifier
    correlation_id: Execution correlation identifier

Returns:
    Tuple of (seeds, status, errors)
    - seeds: Dictionary of generated seeds
    - status: Dictionary of application status
    - errors: List of errors (empty if successful)

Note:
    Errors (including missing mandatory seeds) are reported via the
    returned errors list rather than by raising exceptions.

Specification:
    P00-EN v2.0 Section 3.4 - Determinism Context

Example:
    >>> from orchestration.seed_registry import get_global_seed_registry
    >>> registry = get_global_seed_registry()
    >>> seeds, status, errors = initialize_determinism_from_registry(
    ...     registry, "plan_2024", "exec_001"
    ... )
    >>> assert not errors
    >>> assert status["python"] and status["numpy"]
"""
# Get seeds from registry
seeds = seed_registry.get_seeds_for_context(
    policy_unit_id=policy_unit_id,
    correlation_id=correlation_id
)

# Validate mandatory seeds present
missing = [s for s in MANDATORY_SEEDS if seeds.get(s) is None]
if missing:
    error = f"Missing mandatory seeds from registry: {missing}"
    return seeds, {}, [error]

# Apply seeds to RNGs
try:
    status = apply_seeds_to_rngs(seeds)
except Exception as e:
    return seeds, {}, [f"Failed to apply seeds: {e}"]

# Validate application
success, errors = validate_seed_application(seeds, status)

if not success:
    return seeds, status, errors
    return seeds, status, []


@contextmanager
def deterministic(
    policy_unit_id: str | None = None,
    correlation_id: str | None = None
) -> Iterator[Seeds]:
    """
    Context manager for scoped deterministic execution.

    Seeds Python random and NumPy random based on policy_unit_id and
    correlation_id. Seeds are derived deterministically via SHA-256.

    Args:
        policy_unit_id: Policy unit identifier (default: env var or "default")
        correlation_id: Correlation identifier (default: env var or "run")

    Yields:
        Seeds object with seed values

    Example:
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v1 = random.random()
        ...     a1 = np.random.rand(3)
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v2 = random.random()
        ...     a2 = np.random.rand(3)
        >>> assert v1 == v2  # Deterministic
    """
    base = policy_unit_id or os.getenv("POLICY_UNIT_ID", "default")
    salt = correlation_id or os.getenv("CORRELATION_ID", "run")

    # Derive seeds for mandatory components
    python_seed = derive_seed_from_parts(base, salt, "python")
    numpy_seed = derive_seed_from_parts(base, salt, "numpy")
    quantum_seed = derive_seed_from_parts(base, salt, "quantum")
    neuromorphic_seed = derive_seed_from_parts(base, salt, "neuromorphic")
    meta_learner_seed = derive_seed_from_parts(base, salt, "meta_learner")

    # Apply mandatory seeds
    random.seed(python_seed)
    if NUMPY_AVAILABLE and np is not None:
        np.random.seed(numpy_seed)

    try:
        yield Seeds(
            python=python_seed,
            numpy=numpy_seed,
            quantum=quantum_seed,
            neuromorphic=neuromorphic_seed,
            meta_learner=meta_learner_seed,
        )
    finally:
        pass  # Keep seeded state


def create_deterministic_rng(seed: int) -> Any:
    """
    Create a local deterministic NumPy RNG (doesn't affect global state).

    Args:
        seed: Integer seed

    Returns:
        NumPy Generator instance (or None if NumPy unavailable)

    Example:
        >>> rng = create_deterministic_rng(42)
        >>> if rng is not None:
        ...     v1 = rng.random()
        ...     rng = create_deterministic_rng(42)
        ...     v2 = rng.random()
        ...     assert v1 == v2
    """
    if not NUMPY_AVAILABLE or np is None:
        return None
    return np.random.default_rng(seed)


__all__ = [
    "MANDATORY_SEEDS",
    "OPTIONAL_SEEDS",
    "ALL_SEEDS",
    "Seeds",
    "derive_seed_from_string",
    "derive_seed_from_parts",
    "apply_seeds_to_rngs",
    "validate_seed_application",
    "initialize_determinism_from_registry",
    "deterministic",
    "create_deterministic_rng",
]
```

src/farfan_pipeline/phases/Phase_zero/determinism_helpers.py

```python
"""
Determinism Helpers - Centralized Seeding and State Management
==============================================================

Provides centralized determinism enforcement for the entire pipeline:
- Stable seed derivation from policy_unit_id and correlation_id
- Context manager for scoped deterministic execution
- Controls random, numpy.random, and other stochastic libraries

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

from __future__ import annotations

import json
import os
import random
from contextlib import contextmanager
from dataclasses import dataclass
from hashlib import sha256
from typing import TYPE_CHECKING, Any

import numpy as np

if TYPE_CHECKING:
    from collections.abc import Iterator


def _seed_from(*parts: Any) -> int:
    """
    Derive a 32-bit seed from arbitrary parts via SHA-256.

    Args:
        *parts: Components to hash (will be JSON-serialized)

    Returns:
        32-bit integer seed suitable for random/numpy

    Examples:
        >>> s1 = _seed_from("PU_123", "corr-1")
        >>> s2 = _seed_from("PU_123", "corr-1")
        >>> s1 == s2
        True
        >>> s3 = _seed_from("PU_123", "corr-2")
        >>> s1 != s3
        True
    """
    raw = json.dumps(parts, sort_keys=True, separators=(",", ":"), ensure_ascii=False)
    # 32-bit seed for numpy/py random
    return int(sha256(raw.encode("utf-8")).hexdigest()[:8], 16)
```

```
@dataclass(frozen=True)
class Seeds:
    """Container for seeds used in deterministic execution."""
    py: int
    np: int


@contextmanager
def deterministic(
    policy_unit_id: str | None = None,
    correlation_id: str | None = None
) -> Iterator[Seeds]:
    """
    Context manager for deterministic execution.

    Sets seeds for Python's random and NumPy's random based on
    policy_unit_id and correlation_id. Seeds are derived deterministically
    via SHA-256 hashing.

    Args:
        policy_unit_id: Policy unit identifier (default: env var or "default")
        correlation_id: Correlation identifier (default: env var or "run")

    Yields:
        Seeds object with py and np seed values

    Examples:
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v1 = random.random()
        ...     a1 = np.random.rand(3)
        >>> with deterministic("PU_123", "corr-1") as seeds:
        ...     v2 = random.random()
        ...     a2 = np.random.rand(3)
        >>> v1 == v2  # Deterministic
        True
        >>> np.array_equal(a1, a2)  # Deterministic
        True
    """
    base = policy_unit_id or os.getenv("POLICY_UNIT_ID", "default")
    salt = correlation_id or os.getenv("CORRELATION_ID", "run")
    s = _seed_from("fixed", base, salt)

    # Set seeds for both random modules
    random.seed(s)
    np.random.seed(s)

    try:
        yield Seeds(py=s, np=s)
    finally:
        # Keep deterministic state; caller may reseed per-phase if needed
        pass
```

```
def create_deterministic_rng(seed: int) -> np.random.Generator:
    """
    Create a deterministic NumPy random number generator.

    Use this for local RNG that doesn't affect global state.

    Args:
        seed: Integer seed

    Returns:
        NumPy Generator instance

    Examples:
        >>> rng = create_deterministic_rng(42)
        >>> v1 = rng.random()
        >>> rng = create_deterministic_rng(42)
        >>> v2 = rng.random()
        >>> v1 == v2
        True
    """
    return np.random.default_rng(seed)


if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Integration tests
    print("\n" + "="*60)
    print("Determinism Integration Tests")
    print("="*60)

    print("\n1. Testing seed derivation:")
    s1 = _seed_from("PU_123", "corr-1")
    s2 = _seed_from("PU_123", "corr-1")
    s3 = _seed_from("PU_123", "corr-2")
    assert s1 == s2
    assert s1 != s3
    print(f"   ? Same inputs ? same seed: {s1}")
    print(f"   ? Different inputs ? different seed: {s3}")

    print("\n2. Testing deterministic context with random:")
    with deterministic("PU_123", "corr-1") as seeds1:
        a = random.random()
        b = random.randint(0, 100)
    with deterministic("PU_123", "corr-1") as seeds2:
        c = random.random()
        d = random.randint(0, 100)
    assert a == c
    assert b == d
```

```
print(f"   ? Python random is deterministic: {a:.6f}")
print(f"   ? Python randint is deterministic: {b}")

print("\n3. Testing deterministic context with numpy:")
with deterministic("PU_123", "corr-1") as seeds:
    arr1 = np.random.rand(3).tolist()
with deterministic("PU_123", "corr-1") as seeds:
    arr2 = np.random.rand(3).tolist()
assert arr1 == arr2
print(f"   ? NumPy random is deterministic: {arr1}")

print("\n4. Testing local RNG generator:")
rng1 = create_deterministic_rng(42)
v1 = rng1.random()
rng2 = create_deterministic_rng(42)
v2 = rng2.random()
assert v1 == v2
print(f"   ? Local RNG is deterministic: {v1:.6f}")

print("\n5. Testing different correlation IDs produce different results:")
with deterministic("PU_123", "corr-A"):
    val_a = random.random()
with deterministic("PU_123", "corr-B"):
    val_b = random.random()
assert val_a != val_b
print("   ? Different correlation ? different values")
print(f"      corr-A: {val_a:.6f}")
print(f"      corr-B: {val_b:.6f}")

print("\n" + "="*60)
print("Determinism doctest OK - All tests passed!")
print("="*60)
```

src/farfan_pipeline/phases/Phase_zero/deterministic_execution.py

```
"""
Deterministic Execution Utilities - Production Grade
====================================================

Utilities for ensuring deterministic, reproducible execution across
the policy analysis pipeline.

Features:
- Deterministic random seed management
- UTC-only timestamp handling
- Structured execution logging
- Side-effect isolation
- Reproducible event ID generation

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

import hashlib
import logging
import random
import time
import uuid
from collections.abc import Callable, Iterator
from contextlib import contextmanager
from datetime import datetime, timezone
from typing import Any

import numpy as np

from farfan_pipeline.utils.enhanced_contracts import StructuredLogger, utc_now_iso

# ============================================================================
# DETERMINISTIC SEED MANAGEMENT
# ============================================================================

class DeterministicSeedManager:
    """
    Manages random seeds for deterministic execution.

    All stochastic operations must use seeds managed by this class to ensure
    reproducibility across runs.

    Examples:
        >>> manager = DeterministicSeedManager(base_seed=42)
        >>> with manager.scoped_seed("operation1"):
        ...     value = random.random()
        >>> # Seed is automatically restored after context
    """

    def __init__(self, base_seed: int = 42) -> None:
```

```python
        """
        Initialize seed manager with base seed.

        Args:
            base_seed: Master seed for all derived seeds
        """
        self.base_seed = base_seed
        self._seed_counter = 0
        self._initialize_seeds(base_seed)

    def _initialize_seeds(self, seed: int) -> None:
        """Initialize all random number generators with deterministic seeds."""
        random.seed(seed)
        np.random.seed(seed)
        # For reproducibility, also set hash seed
        # Note: PYTHONHASHSEED should be set in environment for full determinism

    def get_derived_seed(self, operation_name: str) -> int:
        """
        Generate a deterministic seed for a specific operation.

        Args:
            operation_name: Unique name for the operation

        Returns:
            Deterministic integer seed derived from operation name and base seed

        Examples:
            >>> manager = DeterministicSeedManager(42)
            >>> seed1 = manager.get_derived_seed("test")
            >>> seed2 = manager.get_derived_seed("test")
            >>> seed1 == seed2  # Deterministic
            True
        """
        # Use cryptographic hash for stable seed derivation
        hash_input = f"{self.base_seed}:{operation_name}".encode()
        hash_digest = hashlib.sha256(hash_input).digest()
        # Convert first 4 bytes to int
        return int.from_bytes(hash_digest[:4], byteorder='big')

    @contextmanager
    def scoped_seed(self, operation_name: str) -> Iterator[int]:
        """
        Context manager for scoped seed usage.

        Sets seeds for the operation, then restores original state.

        Args:
            operation_name: Unique name for the operation

        Yields:
            Derived seed for this operation

        Examples:
            >>> manager = DeterministicSeedManager(42)
            >>> with manager.scoped_seed("my_operation") as seed:
            ...     result = random.randint(0, 100)
        """
        # Save current state
        random_state = random.getstate()
        np_state = np.random.get_state()

        # Set new seed
        derived_seed = self.get_derived_seed(operation_name)
        self._initialize_seeds(derived_seed)

        try:
            yield derived_seed
        finally:
            # Restore state
            random.setstate(random_state)
            np.random.set_state(np_state)

    def get_event_id(self, operation_name: str, timestamp_utc: str | None = None) ->
str:
        """
        Generate a reproducible event ID for an operation.

        Args:
            operation_name: Operation name
            timestamp_utc: Optional UTC timestamp (ISO-8601); if None, uses current time

        Returns:
            Deterministic event ID based on operation and timestamp

        Examples:
            >>> manager = DeterministicSeedManager(42)
            >>> event_id = manager.get_event_id("test", "2024-01-01T00:00:00Z")
            >>> len(event_id)
            64
        """
        ts = timestamp_utc or utc_now_iso()
        hash_input = f"{self.base_seed}:{operation_name}:{ts}".encode()
        return hashlib.sha256(hash_input).hexdigest()


# ============================================================================
# DETERMINISTIC EXECUTION WRAPPER
# ============================================================================

class DeterministicExecutor:
    """
    Wraps functions to ensure deterministic execution with observability.

    Features:
    - Automatic seed management
    - Structured logging of execution
    - Latency tracking
    - Error handling with event IDs

    Examples:
        >>> executor = DeterministicExecutor(base_seed=42, logger_name="test")
        >>> @executor.deterministic(operation_name="my_func")
        ... def my_function(x: int) -> int:
        ...     return x + random.randint(0, 10)
    """

    def __init__(
        self,
        base_seed: int = 42,
        logger_name: str = "deterministic_executor",
        enable_logging: bool = True
    ) -> None:
        """
        Initialize deterministic executor.

        Args:
            base_seed: Master seed for all operations
            logger_name: Logger name for structured logging
            enable_logging: Whether to enable structured logging
        """
        self.seed_manager = DeterministicSeedManager(base_seed)
        self.logger = StructuredLogger(logger_name) if enable_logging else None
        self.enable_logging = enable_logging

    def deterministic(
        self,
        operation_name: str,
        log_inputs: bool = False,
        log_outputs: bool = False
    ) -> Callable:
        """
        Decorator to make a function deterministic with logging.

        Args:
            operation_name: Unique name for this operation
            log_inputs: Whether to log input parameters
            log_outputs: Whether to log output values

        Returns:
            Decorated function with deterministic execution
        """
        def decorator(func: Callable) -> Callable:
            def wrapper(*args: Any, **kwargs: Any) -> Any:
                # Generate correlation and event IDs
                correlation_id = str(uuid.uuid4())
                event_id = self.seed_manager.get_event_id(operation_name)

                # Start timing
                start_time = time.perf_counter()

                # Execute with scoped seed
```

```
                try:
                    with self.seed_manager.scoped_seed(operation_name) as seed:
                        result = func(*args, **kwargs)

                        # Calculate latency
                        latency_ms = (time.perf_counter() - start_time) * 1000

                        # Log success
                        if self.enable_logging and self.logger:
                            log_data = {
                                "event_id": event_id,
                                "seed": seed,
                                "latency_ms": latency_ms,
                            }
                            if log_inputs:
                                    log_data["inputs"] = str(args)[:100]  # Truncate for
safety
                            if log_outputs:
                                log_data["outputs"] = str(result)[:100]

                            self.logger.log_execution(
                                operation=operation_name,
                                correlation_id=correlation_id,
                                success=True,
                                latency_ms=latency_ms,
                                **log_data
                            )

                        return result

                except Exception as e:
                    # Calculate latency even on error
                    latency_ms = (time.perf_counter() - start_time) * 1000

                    # Log error
                    if self.enable_logging and self.logger:
                        self.logger.log_execution(
                            operation=operation_name,
                            correlation_id=correlation_id,
                            success=False,
                            latency_ms=latency_ms,
                            event_id=event_id,
                            error=str(e)[:200]  # Truncate for safety
                        )

                    # Re-raise with event ID
                        raise RuntimeError(f"[{event_id}] {operation_name} failed: {e}")
from e

            return wrapper
        return decorator


# ============================================================================
# UTC TIMESTAMP UTILITIES
# ============================================================================

def enforce_utc_now() -> datetime:
    """
    Get current UTC datetime.

    Returns:
        Current datetime in UTC timezone

    Examples:
        >>> dt = enforce_utc_now()
        >>> dt.tzinfo is not None
        True
    """
    return datetime.now(timezone.utc)


def parse_utc_timestamp(timestamp_str: str) -> datetime:
    """
    Parse ISO-8601 timestamp and enforce UTC.

    Args:
        timestamp_str: ISO-8601 timestamp string

    Returns:
        Parsed datetime in UTC

    Raises:
        ValueError: If timestamp is not UTC or invalid format

    Examples:
        >>> dt = parse_utc_timestamp("2024-01-01T00:00:00Z")
        >>> dt.year
        2024
    """
    dt = datetime.fromisoformat(timestamp_str.replace('Z', '+00:00'))

    # Enforce UTC
    if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
        raise ValueError(f"Timestamp must be UTC: {timestamp_str}")

    return dt


# ============================================================================
# SIDE-EFFECT ISOLATION
# ============================================================================

@contextmanager
def isolated_execution() -> Iterator[None]:
    """
    Context manager to isolate side effects during execution.
    Current isolation:
    - Prevents print statements (captured and logged as warning)
    - Future: file I/O restrictions, network restrictions

    Yields:
        None

    Examples:
        >>> with isolated_execution():
        ...     # Code here has controlled side effects
        ...     pass
    """
    # For now, minimal isolation - can be extended with more restrictions
    import io
    import sys

    # Capture stdout/stderr to detect violations
    old_stdout = sys.stdout
    old_stderr = sys.stderr
    stdout_capture = io.StringIO()
    stderr_capture = io.StringIO()

    try:
        sys.stdout = stdout_capture
        sys.stderr = stderr_capture
        yield
    finally:
        sys.stdout = old_stdout
        sys.stderr = old_stderr

        # Log any captured output as warning (side effect violation)
        if stdout_capture.getvalue():
            logging.warning(
                "Side effect detected: stdout captured during isolated execution: %s",
                stdout_capture.getvalue()[:200]
            )
        if stderr_capture.getvalue():
            logging.warning(
                "Side effect detected: stderr captured during isolated execution: %s",
                stderr_capture.getvalue()[:200]
            )


# ============================================================================
# IN-SCRIPT TESTS
# ============================================================================

if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)
```

```
# Additional tests
print("\n" + "="*60)
print("Deterministic Execution Tests")
print("="*60)

# Test 1: Seed manager determinism
print("\n1. Testing seed manager determinism:")
manager1 = DeterministicSeedManager(42)
manager2 = DeterministicSeedManager(42)

seed1_a = manager1.get_derived_seed("test_op")
seed1_b = manager1.get_derived_seed("test_op")
seed2_a = manager2.get_derived_seed("test_op")

assert seed1_a == seed1_b == seed2_a, "Seeds must be deterministic"
print(f"   ? Deterministic seeds: {seed1_a} == {seed1_b} == {seed2_a}")

# Test 2: Scoped seed restoration
print("\n2. Testing scoped seed restoration:")
manager = DeterministicSeedManager(42)

initial_value = random.random()
with manager.scoped_seed("temp_operation"):
    _ = random.random()  # Different value inside scope
restored_value = random.random()

# Reset and check if we can reproduce
manager._initialize_seeds(42)
reproduced_value = random.random()

print(f"   ? Initial value: {initial_value:.6f}")
print(f"   ? Reproduced value: {reproduced_value:.6f}")
assert abs(initial_value - reproduced_value) < 1e-10, "Seed restoration failed"
print("   ? Seed restoration successful")

# Test 3: Deterministic executor
print("\n3. Testing deterministic executor:")
executor = DeterministicExecutor(base_seed=42, enable_logging=False)

@executor.deterministic(operation_name="test_function")
def sample_function(n: int) -> float:
    return sum(random.random() for _ in range(n))

result1 = sample_function(5)

# Reset and run again
executor.seed_manager._initialize_seeds(42)
result2 = sample_function(5)

print(f"   ? Result 1: {result1:.6f}")
print(f"   ? Result 2: {result2:.6f}")
assert abs(result1 - result2) < 1e-10, "Deterministic execution failed"
print("   ? Deterministic execution verified")
```

```
# Test 4: UTC enforcement
print("\n4. Testing UTC enforcement:")
utc_now = enforce_utc_now()
print(f"   ? UTC now: {utc_now.isoformat()}")
assert utc_now.tzinfo is not None, "Must have timezone"

# Test 5: Event ID reproducibility
print("\n5. Testing event ID reproducibility:")
manager = DeterministicSeedManager(42)
event_id1 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
event_id2 = manager.get_event_id("operation", "2024-01-01T00:00:00Z")
assert event_id1 == event_id2, "Event IDs must be reproducible"
print(f"   ? Event ID: {event_id1[:16]}...")
print("   ? Event ID reproducibility verified")

print("\n" + "="*60)
print("All tests passed!")
print("="*60)
```

src/farfan_pipeline/phases/Phase_zero/domain_errors.py

```
"""
Domain-Specific Exceptions - Contract Violation Errors
======================================================

Provides domain-specific exception hierarchy for contract violations.

Exception Hierarchy:
    ContractViolationError (base)
    ??? DataContractError (data/payload violations)
    ??? SystemContractError (system/configuration violations)

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""


class ContractViolationError(Exception):
    """
    Base exception for all contract violations.

    Use this as the base class for specific contract violation types.

    Examples:
        >>> try:
        ...     raise ContractViolationError("Contract violated")
        ... except ContractViolationError as e:
        ...     print(f"Caught: {e}")
        Caught: Contract violated
    """
    pass


class DataContractError(ContractViolationError):
    """
    Exception for data/payload contract violations.

    Raised when:
    - Payload schema is invalid
    - Required fields are missing
    - Field values are out of range
    - Data integrity checks fail (e.g., digest mismatch)

    Examples:
        >>> try:
        ...     raise DataContractError("Invalid payload schema")
        ... except DataContractError as e:
        ...     print(f"Data error: {e}")
        Data error: Invalid payload schema
    """
    pass
```

```
class SystemContractError(ContractViolationError):
    """
    Exception for system/configuration contract violations.

    Raised when:
    - System configuration is invalid
    - Required resources are unavailable
    - Environment preconditions are not met
    - Infrastructure failures occur

    Examples:
        >>> try:
        ...     raise SystemContractError("Configuration missing")
        ... except SystemContractError as e:
        ...     print(f"System error: {e}")
        System error: Configuration missing
    """
    pass


if __name__ == "__main__":
    import doctest

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Integration tests
    print("\n" + "="*60)
    print("Domain Exceptions Integration Tests")
    print("="*60)

    print("\n1. Testing exception hierarchy:")
    assert issubclass(DataContractError, ContractViolationError)
    assert issubclass(SystemContractError, ContractViolationError)
    print("   ? DataContractError inherits from ContractViolationError")
    print("   ? SystemContractError inherits from ContractViolationError")

    print("\n2. Testing exception catching:")
    try:
        raise DataContractError("Test data error")
    except ContractViolationError as e:
        assert isinstance(e, DataContractError)
        print("   ? DataContractError caught as ContractViolationError")

    try:
        raise SystemContractError("Test system error")
    except ContractViolationError as e:
        assert isinstance(e, SystemContractError)
        print("   ? SystemContractError caught as ContractViolationError")

    print("\n3. Testing specific exception catching:")
    try:
```

```
    raise DataContractError("Payload validation failed")
except DataContractError as e:
    assert str(e) == "Payload validation failed"
    print("   ? DataContractError caught specifically")

try:
    raise SystemContractError("Config file not found")
except SystemContractError as e:
    assert str(e) == "Config file not found"
    print("   ? SystemContractError caught specifically")

print("\n4. Testing error differentiation:")
errors = []

try:
    raise DataContractError("Data issue")
except ContractViolationError as e:
    errors.append(("data", type(e).__name__))

try:
    raise SystemContractError("System issue")
except ContractViolationError as e:
    errors.append(("system", type(e).__name__))

assert errors[0] == ("data", "DataContractError")
assert errors[1] == ("system", "SystemContractError")
print("   ? Data and system errors are distinguishable")

print("\n" + "="*60)
print("Domain exceptions doctest OK - All tests passed!")
print("="*60)
```

src/farfan_pipeline/phases/Phase_zero/exit_gates.py

```python
"""
Phase 0 Exit Gate Validators
=============================

Implements the 7 strict exit gates defined in P00-EN v2.0 specification (extended).

Exit gates are MANDATORY checkpoints that must pass before proceeding to Phase 1.
Each gate validates a specific aspect of Phase 0 initialization.

Contract:
    Gate 1 (Bootstrap): Runtime config loaded, artifacts dir created
    Gate 2 (Input Verification): PDF and questionnaire hashed
    Gate 3 (Boot Checks): Dependencies validated (PROD: fatal, DEV: warn)
    Gate 4 (Determinism): All required seeds applied to RNGs
    Gate 5 (Questionnaire Integrity): SHA256 validation against known-good
    Gate 6 (Method Registry): Expected method count validation
    Gate 7 (Smoke Tests): Sample methods from major categories

Author: Phase 0 Compliance Team
Version: 2.0.0
Specification: P00-EN v2.0 + P1 Hardening
"""

from __future__ import annotations

import os
from dataclasses import dataclass
from typing import TYPE_CHECKING, Protocol, Any

if TYPE_CHECKING:
    from canonic_phases.Phase_zero.runtime_config import RuntimeConfig


class Phase0Runner(Protocol):
    """Protocol defining the interface for Phase 0 runners."""

    errors: list[str]
    _bootstrap_failed: bool
    runtime_config: RuntimeConfig | None
    seed_snapshot: dict[str, int]
    input_pdf_sha256: str
    questionnaire_sha256: str
    method_executor: Any | None
    questionnaire: Any | None


@dataclass
class GateResult:
    """Result of a Phase 0 exit gate check.

    Attributes:
        passed: True if gate passed, False otherwise
        gate_name: Name of the gate (bootstrap, input_verification, boot_checks, determinism)
        gate_id: Numeric gate ID (1-4)
        reason: Human-readable failure reason (None if passed)
    """

    passed: bool
    gate_name: str
    gate_id: int
    reason: str | None = None

    def to_dict(self) -> dict:
        """Convert to dictionary for logging."""
        return {
            "passed": self.passed,
            "gate_name": self.gate_name,
            "gate_id": self.gate_id,
            "reason": self.reason,
        }


def check_bootstrap_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 1: Bootstrap - Runtime configuration and initialization.

    Validates:
        - Runtime config loaded successfully
        - No bootstrap failures during __init__
        - No errors accumulated during bootstrap

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        Section 3.1 P0.0 - Bootstrap must complete without errors
    """
    gate_id = 1
    gate_name = "bootstrap"

    if runner._bootstrap_failed:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Bootstrap failed during initialization"
        )

    if runner.runtime_config is None:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Runtime config not loaded"
        )

    if runner.errors:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Bootstrap errors detected: {'; '.join(runner.errors)}"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_input_verification_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 2: Input Verification - Cryptographic hashing of inputs.

    Validates:
        - Input PDF exists and is hashed (SHA-256)
        - Questionnaire exists and is hashed (SHA-256)
        - No errors during hashing

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        Section 3.2 P0.1 - Inputs must be cryptographically verified
    """
    gate_id = 2
    gate_name = "input_verification"

    pdf_hash = getattr(runner, "input_pdf_sha256", "") or ""
    if not (
        isinstance(pdf_hash, str)
        and len(pdf_hash) == 64
        and all(c in "0123456789abcdef" for c in pdf_hash.lower())
    ):
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Input PDF not hashed with valid SHA-256"
        )

    questionnaire_hash = getattr(runner, "questionnaire_sha256", "") or ""
    if not (
        isinstance(questionnaire_hash, str)
        and len(questionnaire_hash) == 64
        and all(c in "0123456789abcdef" for c in questionnaire_hash.lower())
    ):
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Questionnaire not hashed with valid SHA-256"
        )

    if runner.errors:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Input verification errors: {'; '.join(runner.errors)}"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_boot_checks_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 3: Boot Checks - Dependency validation.

    Validates:
        - Boot checks executed successfully
        - No errors in PROD mode (DEV mode allows warnings)
        - Critical dependencies available

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        Section 3.3 P0.2 - Boot checks must pass in PROD, warn in DEV

    Note:
        In DEV mode, boot check warnings do NOT populate runner.errors,
        allowing the gate to pass with degraded quality.
    """
    gate_id = 3
    gate_name = "boot_checks"

    if runner.errors:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Boot check errors: {'; '.join(runner.errors)}"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)
```

```
def check_determinism_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 4: Determinism - RNG seeding validation.

    Validates:
        - Seed snapshot created
        - Python seed applied (MANDATORY)
        - NumPy seed applied (MANDATORY)
        - Additional seeds present for advanced components
        - No errors during seeding

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        Section 3.4 P0.3 - Deterministic seeds must be applied

    Critical Seeds:
        - python: Python random module (MANDATORY)
        - numpy: NumPy random state (MANDATORY)
        - quantum: Quantum optimizer (optional if unused)
        - neuromorphic: Neuromorphic controller (optional if unused)
        - meta_learner: Meta-learner strategy (optional if unused)
    """
    gate_id = 4
    gate_name = "determinism"

    if not hasattr(runner, 'seed_snapshot'):
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Seed snapshot not created"
        )

    # MANDATORY seeds (system cannot run without these)
    MANDATORY_SEEDS = ["python", "numpy"]
     missing_mandatory = [s for s in MANDATORY_SEEDS if runner.seed_snapshot.get(s) is
None]

    if missing_mandatory:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Missing mandatory seeds: {missing_mandatory}"
        )

    if runner.errors:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Determinism errors: {'; '.join(runner.errors)}"
        )

    # OPTIONAL seeds (log warning if missing, but don't fail gate)
    OPTIONAL_SEEDS = ["quantum", "neuromorphic", "meta_learner"]
    missing_optional = [s for s in OPTIONAL_SEEDS if runner.seed_snapshot.get(s) is
None]

    if missing_optional:
        # Don't fail gate, but note in reason for observability
        return GateResult(
            passed=True,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Optional seeds missing (non-fatal): {missing_optional}"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_questionnaire_integrity_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 5: Questionnaire Integrity - SHA256 validation against known-good.

    Validates:
        - Questionnaire SHA256 hash computed correctly
        - Hash matches expected/configured value (from env or RuntimeConfig)
        - No corruption detected in questionnaire data

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        P1 Hardening - Questionnaire must match cryptographic fingerprint

    Note:
        Expected hash can be set via:
        - Environment variable: EXPECTED_QUESTIONNAIRE_SHA256
        - RuntimeConfig.expected_questionnaire_sha256
        - If not set, gate passes with warning (legacy compatibility)
    """
    gate_id = 5
    gate_name = "questionnaire_integrity"

    # Get expected hash from environment or RuntimeConfig
    expected_hash = os.getenv("EXPECTED_QUESTIONNAIRE_SHA256", "").strip()

    if runner.runtime_config and hasattr(runner.runtime_config,
"expected_questionnaire_sha256"):
           config_hash = getattr(runner.runtime_config, "expected_questionnaire_sha256",
"")
        if config_hash:
            expected_hash = config_hash

    # If no expected hash configured, pass with warning (legacy mode)
    if not expected_hash:
        return GateResult(
            passed=True,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="No expected questionnaire hash configured (legacy mode)"
        )

    # Validate format of expected hash
    if not (
        isinstance(expected_hash, str)
        and len(expected_hash) == 64
        and all(c in "0123456789abcdef" for c in expected_hash.lower())
    ):
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Invalid expected hash format: {expected_hash[:16]}..."
        )

    # Get actual questionnaire hash
    actual_hash = getattr(runner, "questionnaire_sha256", "")

    if not actual_hash:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="Questionnaire hash not computed"
        )

    # Compare hashes (case-insensitive)
    if actual_hash.lower() != expected_hash.lower():
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
             reason=f"Questionnaire hash mismatch: expected {expected_hash[:16]}..., got
{actual_hash[:16]}..."
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_method_registry_gate(runner: Phase0Runner) -> GateResult:
    """
    Gate 6: Method Registry - Expected method count validation.

    Validates:
        - MethodRegistry/MethodExecutor is available
        - Expected number of methods are registered and loadable
        - Method registry statistics are accessible

    Args:
        runner: Phase 0 runner instance

    Returns:
        GateResult with pass/fail status

    Specification:
        P1 Hardening - All expected methods must be loadable

    Critical Thresholds:
        - EXPECTED_METHOD_COUNT from environment (default: 416)
        - Registered classes must match expected count
        - Failed classes count must be zero in PROD mode
    """
    gate_id = 6
    gate_name = "method_registry"

    # Get expected method count from environment or RuntimeConfig
    expected_count = int(os.getenv("EXPECTED_METHOD_COUNT", "416"))

    if runner.runtime_config and hasattr(runner.runtime_config, "expected_method_count"):
        config_count = getattr(runner.runtime_config, "expected_method_count", None)
        if config_count:
            expected_count = config_count

    # Check if method executor is available
    method_executor = getattr(runner, "method_executor", None)

    if method_executor is None:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="MethodExecutor not initialized"
        )

    # Get method registry from executor
    method_registry = None
    if hasattr(method_executor, "_method_registry"):
        method_registry = method_executor._method_registry
    elif hasattr(method_executor, "method_registry"):
        method_registry = method_executor.method_registry

    if method_registry is None:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason="MethodRegistry not accessible from MethodExecutor"
        )

    # Get registry statistics
    try:
        stats = method_registry.get_stats()
    except Exception as exc:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Failed to get registry stats: {exc}"
        )

    # Validate method count
    registered_count = stats.get("total_classes_registered", 0)
    failed_count = stats.get("failed_classes", 0)

    if registered_count < expected_count:
        return GateResult(
            passed=False,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"Method count mismatch: expected {expected_count}, registered {registered_count}"
        )

    # In PROD mode, no failed classes allowed
    if runner.runtime_config and hasattr(runner.runtime_config, "mode"):
        from canonic_phases.Phase_zero.runtime_config import RuntimeMode
        if runner.runtime_config.mode == RuntimeMode.PROD and failed_count > 0:
            failed_names = stats.get("failed_class_names", [])
            return GateResult(
                passed=False,
                gate_name=gate_name,
                gate_id=gate_id,
                reason=f"PROD mode: {failed_count} failed classes: {failed_names[:3]}"
            )

    # Pass with warning if failed classes in DEV mode
    if failed_count > 0:
        return GateResult(
            passed=True,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"DEV mode: {failed_count} failed classes (non-fatal)"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_smoke_tests_gate(runner: Phase0Runner) -> GateResult:
```

```
"""
Gate 7: Smoke Tests - Sample methods from major categories.

Validates:
    - Ingest category: Sample method can be instantiated
    - Scoring category: Sample method can be instantiated
    - Aggregation category: Sample method can be instantiated

Args:
    runner: Phase 0 runner instance

Returns:
    GateResult with pass/fail status

Specification:
    P1 Hardening - Critical method categories must be operational

Smoke Test Categories:
    - Ingest: PDFChunkExtractor or similar (Phase 1 dependency)
    - Scoring: SignalEnrichedScorer or similar (Phase 3 dependency)
    - Aggregation: DimensionAggregator or similar (Phase 4 dependency)
"""
gate_id = 7
gate_name = "smoke_tests"

method_executor = getattr(runner, "method_executor", None)

if method_executor is None:
    return GateResult(
        passed=False,
        gate_name=gate_name,
        gate_id=gate_id,
        reason="MethodExecutor not available for smoke tests"
    )

# Define smoke test samples (class_name, category)
smoke_tests = [
    ("PDFChunkExtractor", "ingest"),
    ("SemanticAnalyzer", "scoring"),
    ("DimensionAggregator", "aggregation"),
]

failed_tests = []

for class_name, category in smoke_tests:
    try:
        # Check if method exists in registry
        if hasattr(method_executor, "instances"):
            # Try to access instance (will instantiate if not cached)
            if hasattr(method_executor.instances, "get"):
                instance = method_executor.instances.get(class_name)
                if instance is None:
                    failed_tests.append(f"{category}:{class_name}")
        else:
                # Fallback: try has_method
                if hasattr(method_executor, "has_method"):
                    # Pick a common method name to check
                    if not method_executor.has_method(class_name, "__init__"):
                        failed_tests.append(f"{category}:{class_name}")
        except Exception as exc:
            failed_tests.append(f"{category}:{class_name}({type(exc).__name__})")

    if failed_tests:
        # In PROD mode, any smoke test failure is fatal
        if runner.runtime_config and hasattr(runner.runtime_config, "mode"):
            from canonic_phases.Phase_zero.runtime_config import RuntimeMode
            if runner.runtime_config.mode == RuntimeMode.PROD:
                return GateResult(
                    passed=False,
                    gate_name=gate_name,
                    gate_id=gate_id,
                    reason=f"Smoke tests failed: {', '.join(failed_tests)}"
                )

        # In DEV mode, pass with warning
        return GateResult(
            passed=True,
            gate_name=gate_name,
            gate_id=gate_id,
            reason=f"DEV mode: smoke tests failed (non-fatal): {',
'.join(failed_tests)}"
        )

    return GateResult(passed=True, gate_name=gate_name, gate_id=gate_id)


def check_all_gates(runner: Phase0Runner) -> tuple[bool, list[GateResult]]:
    """
    Check all 7 Phase 0 exit gates in sequence.

    Gates are checked in order:
        1. Bootstrap
        2. Input Verification
        3. Boot Checks
        4. Determinism
        5. Questionnaire Integrity (NEW - P1 Hardening)
        6. Method Registry (NEW - P1 Hardening)
        7. Smoke Tests (NEW - P1 Hardening)

    If any gate fails, subsequent gates are NOT checked (fail-fast).

    Args:
        runner: Phase 0 runner instance

    Returns:
        Tuple of (all_passed, results)
        - all_passed: True only if all gates passed
        - results: List of GateResult objects (may be incomplete if fail-fast)
    Example:
        >>> all_passed, results = check_all_gates(runner)
        >>> if not all_passed:
        ...     failed_gate = next(r for r in results if not r.passed)
        ...     print(f"Gate {failed_gate.gate_id} failed: {failed_gate.reason}")
    """
    gates = [
        check_bootstrap_gate,
        check_input_verification_gate,
        check_boot_checks_gate,
        check_determinism_gate,
        check_questionnaire_integrity_gate,
        check_method_registry_gate,
        check_smoke_tests_gate,
    ]

    results = []
    for gate_func in gates:
        result = gate_func(runner)
        results.append(result)

        if not result.passed:
            # Fail-fast: don't check remaining gates
            return False, results

    return True, results


def get_gate_summary(results: list[GateResult]) -> str:
    """
    Generate human-readable summary of gate results.

    Args:
        results: List of GateResult objects from check_all_gates()

    Returns:
        Formatted summary string

    Example:
        >>> _, results = check_all_gates(runner)
        >>> print(get_gate_summary(results))
        Phase 0 Exit Gates: 7/7 passed
          ? Gate 1 (bootstrap): PASS
          ? Gate 2 (input_verification): PASS
          ? Gate 3 (boot_checks): PASS
          ? Gate 4 (determinism): PASS
          ? Gate 5 (questionnaire_integrity): PASS
          ? Gate 6 (method_registry): PASS
          ? Gate 7 (smoke_tests): PASS
    """
    passed = sum(1 for r in results if r.passed)
    total = 7  # There are now 7 Phase 0 gates (4 original + 3 new)
    lines = [f"Phase 0 Exit Gates: {passed}/{total} passed"]

    for result in results:
        status = "?" if result.passed else "?"
        gate_desc = f"Gate {result.gate_id} ({result.gate_name})"

        if result.passed:
            if result.reason:
                # Passed with warning
                lines.append(f"  {status} {gate_desc}: PASS (??  {result.reason})")
            else:
                lines.append(f"  {status} {gate_desc}: PASS")
        else:
            lines.append(f"  {status} {gate_desc}: FAIL - {result.reason}")

    return "\n".join(lines)


__all__ = [
    "Phase0Runner",
    "GateResult",
    "check_bootstrap_gate",
    "check_input_verification_gate",
    "check_boot_checks_gate",
    "check_determinism_gate",
    "check_questionnaire_integrity_gate",
    "check_method_registry_gate",
    "check_smoke_tests_gate",
    "check_all_gates",
    "get_gate_summary",
]
```

src/farfan_pipeline/phases/Phase_zero/hash_utils.py

```
"""
Hash utilities for deterministic content hashing.

This module provides cryptographic hashing functions used across the pipeline
for content integrity verification and change detection.

Author: Integration Team
Version: 1.0.0
Python: 3.10+
"""

import hashlib
import json
from typing import Any


def compute_hash(data: dict[str, Any]) -> str:
    """
    Compute deterministic SHA-256 hash of dictionary data.

    This function creates a canonical JSON representation with sorted keys
    and stable separators to ensure identical dictionaries always produce
    the same hash, regardless of key insertion order.

    Args:
        data: Dictionary to hash

    Returns:
        Hexadecimal SHA-256 digest (64 characters)

    Example:
        >>> data = {"b": 2, "a": 1}
        >>> hash1 = compute_hash(data)
        >>> hash2 = compute_hash({"a": 1, "b": 2})
        >>> hash1 == hash2
        True
    """
    canonical_json = json.dumps(
        data, sort_keys=True, ensure_ascii=True, separators=(",", ":")
    )
    return hashlib.sha256(canonical_json.encode("utf-8")).hexdigest()
```

src/farfan_pipeline/phases/Phase_zero/json_logger.py

```
"""
Lightweight JSON Logging - Structured Event Logging
===================================================

Provides structured JSON logging for the pipeline with:
- JSON formatter for LogRecord
- Helper for logging I/O events with envelope metadata
- No PII logging
- Correlation ID and event ID tracking

Author: Policy Analytics Research Unit
Version: 1.0.0
License: Proprietary
"""

from __future__ import annotations

import json
import logging
import time
from typing import Any

# Import will be available at runtime
try:
    from farfan_pipeline.utils.contract_io import ContractEnvelope
except ImportError:
    # Allow module to load for testing
    ContractEnvelope = None  # type: ignore


class JsonFormatter(logging.Formatter):
    """
    JSON formatter for structured logging.

    Formats LogRecord as JSON with standard fields plus custom extras.
    """

    def format(self, record: logging.LogRecord) -> str:
        """
        Format LogRecord as JSON string.

        Args:
            record: LogRecord to format

        Returns:
            JSON string representation
        """
        payload: dict[str, Any] = {
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "timestamp_utc": record.__dict__.get("timestamp_utc"),
            "event_id": record.__dict__.get("event_id"),
            "correlation_id": record.__dict__.get("correlation_id"),
            "policy_unit_id": record.__dict__.get("policy_unit_id"),
            "phase": record.__dict__.get("phase"),
            "latency_ms": record.__dict__.get("latency_ms"),
            "input_bytes": record.__dict__.get("input_bytes"),
            "output_bytes": record.__dict__.get("output_bytes"),
            "input_digest": record.__dict__.get("input_digest"),
            "output_digest": record.__dict__.get("output_digest"),
        }
        # Drop None values to keep JSON compact
        payload = {k: v for k, v in payload.items() if v is not None}
        return json.dumps(payload, separators=(",", ":"), ensure_ascii=False)


def get_json_logger(name: str = "farfan_core") -> logging.Logger:
    """
    Get or create a JSON logger.

    Creates a logger with JSON formatting if not already configured.

    Args:
        name: Logger name

    Returns:
        Configured logger instance

    Examples:
        >>> logger = get_json_logger("test")
        >>> logger.name
        'test'
        >>> logger.level
        20
    """
    logger = logging.getLogger(name)
    if not any(isinstance(h, logging.StreamHandler) for h in logger.handlers):
        h = logging.StreamHandler()
        h.setFormatter(JsonFormatter())
        logger.addHandler(h)
        logger.setLevel(logging.INFO)
        logger.propagate = False
    return logger


def log_io_event(
    logger: logging.Logger,
    *,
    phase: str,
    envelope_in: Any | None,  # ContractEnvelope or None
    envelope_out: Any,  # ContractEnvelope
    started_monotonic: float,
) -> None:
    """
    Log an I/O event with envelope metadata.
```

```
Args:
    logger: Logger instance
    phase: Phase name
    envelope_in: Input envelope (may be None)
    envelope_out: Output envelope
    started_monotonic: Monotonic start time

Examples:
    >>> import time
    >>> from farfan_core.utils.contract_io import ContractEnvelope
    >>> logger = get_json_logger("test")
    >>> out = ContractEnvelope.wrap(
    ...     {"ok": True},
    ...     policy_unit_id="PU_123",
    ...     correlation_id="corr-1"
    ... )
    >>> # This will log JSON to stdout
    >>> log_io_event(
    ...     logger,
    ...     phase="normalize",
    ...     envelope_in=None,
    ...     envelope_out=out,
    ...     started_monotonic=time.monotonic()
    ... )  # doctest: +SKIP
"""
elapsed_ms = int((time.monotonic() - started_monotonic) * 1000)

# Safely get payload sizes
input_bytes = None
if envelope_in is not None:
    try:
        payload = getattr(envelope_in, "payload", None)
        if payload is not None:
            input_bytes = len(json.dumps(payload, ensure_ascii=False))
    except (TypeError, AttributeError):
        pass

output_bytes = None
try:
    output_bytes = len(json.dumps(envelope_out.payload, ensure_ascii=False))
except (TypeError, AttributeError):
    # If payload is missing or not serializable, skip logging output_bytes.
    # This is non-critical for logging; output_bytes will be None.
    pass

logger.info(
    "phase_io",
    extra={
        "timestamp_utc": envelope_out.timestamp_utc,
        "event_id": envelope_out.event_id,
        "correlation_id": envelope_out.correlation_id,
        "policy_unit_id": envelope_out.policy_unit_id,
        "phase": phase,
            "latency_ms": elapsed_ms,
            "input_bytes": input_bytes,
            "output_bytes": output_bytes,
            "input_digest": getattr(envelope_in, "content_digest", None),
            "output_digest": envelope_out.content_digest,
        },
    )


if __name__ == "__main__":
    import doctest
    import time

    # Run doctests
    print("Running doctests...")
    doctest.testmod(verbose=True)

    # Integration tests
    print("\n" + "="*60)
    print("JSON Logger Integration Tests")
    print("="*60)

    print("\n1. Testing JSON formatter:")
    logger = get_json_logger("demo")
    assert logger.level == logging.INFO
    assert len(logger.handlers) > 0
    assert isinstance(logger.handlers[0].formatter, JsonFormatter)
    print("   ? Logger configured with JSON formatter")

    print("\n2. Testing log output structure:")
    # Create a test record
    record = logging.LogRecord(
        name="test",
        level=logging.INFO,
        pathname="",
        lineno=0,
        msg="test message",
        args=(),
        exc_info=None,
    )
    record.event_id = "evt-123"
    record.correlation_id = "corr-456"
    record.latency_ms = 42

    formatter = JsonFormatter()
    output = formatter.format(record)
    parsed = json.loads(output)

    assert parsed["level"] == "INFO"
    assert parsed["message"] == "test message"
    assert parsed["event_id"] == "evt-123"
    assert parsed["correlation_id"] == "corr-456"
    assert parsed["latency_ms"] == 42
    print("   ? JSON format includes all expected fields")
    print("\n3. Testing I/O event logging:")
    # Only test if ContractEnvelope is available
    if ContractEnvelope is not None:
        from farfan_pipeline.utils.contract_io import ContractEnvelope

        lg = get_json_logger("demo")
        out = ContractEnvelope.wrap(
            {"ok": True},
            policy_unit_id="PU_123",
            correlation_id="corr-1"
        )

        # Capture the log output
        import io
        import sys
        old_stdout = sys.stdout
        sys.stdout = buffer = io.StringIO()

        log_io_event(
            lg,
            phase="normalize",
            envelope_in=None,
            envelope_out=out,
            started_monotonic=time.monotonic()
        )

        sys.stdout = old_stdout
        log_output = buffer.getvalue()

        # Verify JSON output
        if log_output.strip():
            log_data = json.loads(log_output.strip())
            assert log_data["phase"] == "normalize"
            assert log_data["policy_unit_id"] == "PU_123"
            assert "latency_ms" in log_data
            print("   ? I/O event logged with correct structure")
        else:
            print("   ? I/O event logging executed (output suppressed)")
    else:
        print("   ? Skipped (ContractEnvelope not available)")

    print("\n" + "="*60)
    print("JSON logger doctest OK - All tests passed!")
    print("="*60)
```

src/farfan_pipeline/phases/Phase_zero/main.py

```
#!/usr/bin/env python3
"""
F.A.R.F.A.N Verified Pipeline Runner
=====================================

Framework for Advanced Retrieval of Administrativa Narratives

Canonical entrypoint for executing the F.A.R.F.A.N policy analysis pipeline with
cryptographic verification and structured claim logging. This script is designed
to be machine-auditable and produces verifiable artifacts at every step.

Key Features:
- Computes SHA256 hashes of all inputs and outputs
- Emits structured JSON claims for all operations
- Generates verification_manifest.json with success status
- Enforces zero-trust validation principles
- No fabricated logs or unverifiable banners

Usage:
    python -m farfan_core.scripts.run_policy_pipeline_verified [--plan PLAN_PDF]

Requirements:
    - Input PDF must exist (default: data/plans/Plan_1.pdf)
    - Package installed via ``pip install -e .``
    - Write access to artifacts/ directory
"""

from __future__ import annotations

import asyncio
import hashlib
import json
import os
import platform
import random
import sys
import time
import traceback
from dataclasses import asdict, dataclass
from datetime import datetime
from pathlib import Path
from typing import Any, Dict, List, Optional

import farfan_pipeline
from canonic_phases.Phase_zero.paths import PROJECT_ROOT

if os.environ.get("PIPELINE_DEBUG"):
    print(f"DEBUG: farfan_pipeline loaded from {farfan_pipeline.__file__}", flush=True)

# Import contract enforcement infrastructure
from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, get_runtime_config
from canonic_phases.Phase_zero.boot_checks import (
    run_boot_checks,
    get_boot_check_summary,
    BootCheckError,
)
from farfan_pipeline.core.observability.structured_logging import (
    log_runtime_config_loaded,
)
from orchestration.seed_registry import get_global_seed_registry
from orchestration.verification_manifest import (
    VerificationManifest as VerificationManifestBuilder,
    verify_manifest_integrity,
)
from farfan_pipeline.core.phases.phase2_types import validate_phase2_result
from orchestration.versions import get_all_versions


@dataclass
class ExecutionClaim:
    """Structured claim about a pipeline operation."""

    timestamp: str
    claim_type: str  # "start", "complete", "error", "artifact", "hash"
    component: str
    message: str
    data: Optional[Dict[str, Any]] = None

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary for JSON serialization."""
        return asdict(self)


@dataclass
class VerificationManifest:
    """Complete verification manifest for pipeline execution."""

    success: bool
    execution_id: str
    start_time: str
    end_time: str
    input_pdf_path: str
    input_pdf_sha256: str
    artifacts_generated: List[str]
    artifact_hashes: Dict[str, str]
    phases_completed: int
    phases_failed: int
    total_claims: int
    errors: List[str]

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary for JSON serialization."""
        return asdict(self)


class VerifiedPipelineRunner:
    """Executes pipeline with cryptographic verification and claim logging."""

    def __init__(
        self,
        plan_pdf_path: Path,
        artifacts_dir: Path,
        questionnaire_path: Optional[Path] = None,
    ):
        """
        Initialize verified runner.

        Args:
            plan_pdf_path: Path to input PDF
            artifacts_dir: Directory for output artifacts
            questionnaire_path: Optional path to questionnaire file.
                                                  If  None,  uses  canonical  path  from
farfan_core.config.paths.QUESTIONNAIRE_FILE
        """
        self.plan_pdf_path = plan_pdf_path
        self.artifacts_dir = artifacts_dir
        self.claims: List[ExecutionClaim] = []
        self.execution_id = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        self.start_time = datetime.utcnow().isoformat()
        self.phases_completed = 0
        self.phases_failed = 0
        self.errors: List[str] = []
        self.policy_unit_id = f"policy_unit::{self.plan_pdf_path.stem}"
        self.correlation_id = self.execution_id
        self.versions = get_all_versions()
        self.phase2_report: dict[str, Any] | None = None
        self.phase2_metrics: dict[str, Any] | None = None
        self._last_manifest_success: bool = False
        self._bootstrap_failed: bool = False

        # Set questionnaire path (explicit input, SIN_CARRETA compliance)
        if questionnaire_path is None:
            from canonic_phases.Phase_zero.paths import QUESTIONNAIRE_FILE

            questionnaire_path = QUESTIONNAIRE_FILE

        self.questionnaire_path = questionnaire_path

        # Initialize seed registry for deterministic execution
        self.seed_registry = get_global_seed_registry()
        self.seed_snapshot = self._initialize_determinism_context()

        # Initialize verification manifest builder
        manifest_secret = os.getenv("VERIFICATION_HMAC_SECRET") or os.getenv(
            "MANIFEST_SECRET_KEY"
        )
        self.manifest_builder = VerificationManifestBuilder(hmac_secret=manifest_secret)
        self.manifest_builder.manifest_data["versions"] = dict(self.versions)

        # Initialize path and import policies
        try:
            from farfan_pipeline.observability.policy_builder import (
                compute_repo_root,
                build_import_policy,
                build_path_policy,
            )

            self.repo_root = compute_repo_root()
            self.import_policy = build_import_policy(self.repo_root)
            self.path_policy = build_path_policy(self.repo_root)
            self.path_import_report = None
        except Exception as e:
            self.log_claim(
                "error", "policy_init", f"Failed to initialize policies: {e}"
            )
            self.errors.append(f"Failed to initialize policies: {e}")
            self._bootstrap_failed = True
            self.path_import_report = None

        # Ensure artifacts directory exists
        try:
            self.artifacts_dir.mkdir(parents=True, exist_ok=True)
        except Exception as e:
            self.log_claim(
                "error", "bootstrap", f"Failed to create artifacts directory: {e}"
            )
            self.errors.append(f"Failed to create artifacts directory: {e}")
            self._bootstrap_failed = True

        # Initialize runtime configuration
        self.runtime_config: Optional[RuntimeConfig] = None
        try:
            self.runtime_config = RuntimeConfig.from_env()
            self.log_claim(
                "start",
                "runtime_config",
                f"Runtime configuration loaded: {self.runtime_config}",
                {
                    "mode": self.runtime_config.mode.value,
                    "strict_mode": self.runtime_config.is_strict_mode(),
                },
            )

            # Log runtime config for observability
            log_runtime_config_loaded(
                config_repr=repr(self.runtime_config),
                runtime_mode=self.runtime_config.mode,
            )
        except Exception as e:
            self.log_claim(
                "error", "runtime_config", f"Failed to load runtime config: {e}"
            )
            self.errors.append(f"Failed to load runtime config: {e}")
            self._bootstrap_failed = True
            self.runtime_config = None

        # Log bootstrap complete claim
        if not self._bootstrap_failed:
            self.log_claim(
                "start",
                "bootstrap",
                "Bootstrap complete",
                {
                    "execution_id": self.execution_id,
                    "policy_unit_id": self.policy_unit_id,
                    "plan_pdf_path": str(self.plan_pdf_path),
                    "questionnaire_path": str(self.questionnaire_path),
                    "versions": dict(self.versions),
                },
            )

    def _initialize_determinism_context(self) -> dict[str, int]:
        """
        Seed all deterministic sources (python, numpy, etc.) via SeedRegistry.

        Returns:
            Snapshot of generated seeds keyed by component.
        """
        seeds = self.seed_registry.get_seeds_for_context(
            policy_unit_id=self.policy_unit_id,
            correlation_id=self.correlation_id,
        )

        python_seed = seeds.get("python")
        if python_seed is not None:
            random.seed(python_seed)
        else:
            self.log_claim(
                "error", "determinism", "Missing python seed in registry response"
            )
            self.errors.append("Missing python seed in registry response")
            self._bootstrap_failed = True

        numpy_seed = seeds.get("numpy")
        if numpy_seed is not None:
            try:
                import numpy as np

                np.random.seed(numpy_seed)
            except Exception as exc:
                self.log_claim(
                    "warning",
                    "determinism",
                    f"Failed to seed NumPy RNG: {exc}",
                    {"seed": numpy_seed},
                )

        if not self._bootstrap_failed:
            self.log_claim(
                "start",
                "determinism",
                "Deterministic seeds applied",
                {
                    "seeds": seeds,
                    "policy_unit_id": self.policy_unit_id,
                    "correlation_id": self.correlation_id,
                },
            )

        return seeds

    def log_claim(
        self,
        claim_type: str,
        component: str,
        message: str,
        data: Optional[Dict[str, Any]] = None,
    ) -> None:
        """
        Log a structured claim.

        Args:
            claim_type: Type of claim (start, complete, error, artifact, hash)
            component: Component making the claim
            message: Human-readable message
            data: Optional structured data
        """
        claim = ExecutionClaim(
            timestamp=datetime.utcnow().isoformat(),
            claim_type=claim_type,
            component=component,
            message=message,
            data=data or {},
        )
        self.claims.append(claim)

        # Also print for real-time monitoring
        claim_json = json.dumps(claim.to_dict(), separators=(",", ":"))
        print(f"CLAIM: {claim_json}", flush=True)

    def compute_sha256(self, file_path: Path) -> str:
        """
        Compute SHA256 hash of a file.

        Args:
            file_path: Path to file

        Returns:
            Hex-encoded SHA256 hash
        """
        sha256_hash = hashlib.sha256()
        with open(file_path, "rb") as f:
            for byte_block in iter(lambda: f.read(4096), b""):
                sha256_hash.update(byte_block)
        return sha256_hash.hexdigest()

    def _verify_and_hash_file(
        self, file_path: Path, file_type: str, attr_name: str
    ) -> bool:
        """
        Verify file exists and compute its SHA256 hash.

        Args:
            file_path: Path to file to verify and hash
            file_type: Human-readable file type (e.g., "Input PDF", "Questionnaire")
            attr_name: Attribute name to store hash (e.g., "input_pdf_sha256")

        Returns:
            True if verification successful, False otherwise
        """
        # Verify file exists
        if not file_path.exists():
            error_msg = f"{file_type} not found: {file_path}"
            self.log_claim("error", "input_verification", error_msg)
            self.errors.append(error_msg)
            return False

        # Compute hash
        try:
            file_hash = self.compute_sha256(file_path)
            setattr(self, attr_name, file_hash)
            self.log_claim(
                "hash",
                "input_verification",
                f"{file_type} SHA256: {file_hash}",
                {"file": str(file_path), "hash": file_hash},
            )
            return True
        except Exception as e:
            error_msg = f"Failed to hash {file_type}: {str(e)}"
            self.log_claim("error", "input_verification", error_msg)
            self.errors.append(error_msg)
            return False

    def verify_input(self) -> bool:
        """
        Verify input PDF and questionnaire exist and compute hashes.

        Returns:
            True if all inputs are valid
        """
        self.log_claim(
            "start", "input_verification", "Verifying input files (PDF + questionnaire)"
        )

        # Verify and hash PDF
        if not self._verify_and_hash_file(
            self.plan_pdf_path, "Input PDF", "input_pdf_sha256"
        ):
            return False

        # Verify and hash questionnaire (CRITICAL for SIN_CARRETA compliance)
        if not self._verify_and_hash_file(
            self.questionnaire_path, "Questionnaire", "questionnaire_sha256"
        ):
            return False

        self.log_claim(
            "complete",
            "input_verification",
            "Input verification successful (PDF + questionnaire)",
            {
                "pdf_path": str(self.plan_pdf_path),
                "questionnaire_path": str(self.questionnaire_path),
            },
        )
        return True

    def run_boot_checks(self) -> bool:
        """
        Run boot-time validation checks.

        Returns:
            True if all checks pass or fallbacks are allowed

        Raises:
            BootCheckError: If critical check fails in PROD mode
        """
        self.log_claim("start", "boot_checks", "Running boot-time validation checks")

        try:
            results = run_boot_checks(self.runtime_config)
            summary = get_boot_check_summary(results)

            # Log summary
            self.log_claim(
                "complete",
                "boot_checks",
                f"Boot checks completed\n{summary}",
                {"results": results},
            )

            # Print summary for visibility
            print("\n" + summary + "\n", flush=True)

            return True

        except BootCheckError as e:
            error_msg = f"Boot check failed: {e}"
            # In PROD mode, this is fatal
            if self.runtime_config.mode.value == "prod":
                self.log_claim(
                    "error",
                    "boot_checks",
                    error_msg,
                    {"component": e.component, "code": e.code, "reason": e.reason},
                )
                self.errors.append(error_msg)
                print(f"\n? FATAL: {error_msg}\n", flush=True)
                raise

            # In DEV/EXPLORATORY, log warning but continue
            # CRITICAL: Do NOT append to self.errors if we intend to continue,
            # as Phase 0 exit condition requires self.errors to be empty.
            self.log_claim(
                "warning",
                "boot_checks",
                error_msg,
                {"component": e.component, "code": e.code, "reason": e.reason},
            )

            print(
                                       f"\n?? WARNING: {error_msg} (continuing in
{self.runtime_config.mode.value} mode)\n",
                flush=True,
            )
            return False

    async def run(self) -> bool:
        """
        Execute the complete verified pipeline.

        Returns:
            True if pipeline succeeded, False otherwise
        """
        # Check for bootstrap failures (Phase 0.0)
        if self._bootstrap_failed or self.errors:
            self.generate_verification_manifest([], {})
            return False

        self.log_claim("start", "pipeline", "Starting verified pipeline execution")

        # Step 1: Verify input
        if not self.verify_input():
            self.generate_verification_manifest([], {})
            return False

        # STRICT PHASE 0 EXIT GATE: Input Verification
        if self.errors:
            self.log_claim(
                "error",
                "phase0_gate",
                "Phase 0 failure: Errors detected after input verification",
            )
            self.generate_verification_manifest([], {})
            return False

        # Step 1.5: Run boot checks
        try:
            # Ensure runtime_config is available (should be if bootstrap passed, but be safe)
            if self.runtime_config is None:
                raise BootCheckError(
                    "Runtime config is None",
                    "BOOT_CONFIG_MISSING",
                    "Runtime config not initialized",
                )

            if not self.run_boot_checks():
                # Boot checks failed but we're in DEV mode - log warning
                self.log_claim(
                    "warning",
                    "boot_checks",
                    "Boot checks failed but continuing in non-PROD mode",
                )
        except BootCheckError:
            # Boot check failed in PROD mode - abort
            self.generate_verification_manifest([], {})
            return False

        # STRICT PHASE 0 EXIT GATE: Boot Checks
        # If run_boot_checks returned False (Dev mode warning), self.errors should be empty.
        # If it raised (Prod mode), we caught it and returned False above.
        # If any other errors accumulated, abort.
        if self.errors:
            self.log_claim(
                "error",
                "phase0_gate",
                "Phase 0 failure: Errors detected after boot checks",
            )
            self.generate_verification_manifest([], {})
            return False

        # Step 1.75: Run path and import verification
        self.log_claim(
            "start", "path_import_verification", "Running path and import verification"
        )

        try:
            from farfan_pipeline.observability.import_scanner import validate_imports
            from farfan_pipeline.observability.path_guard import guard_paths_and_imports
            from farfan_pipeline.observability.path_import_policy import (
                PolicyReport,
                merge_policy_reports,
            )
            # Static import analysis
            static_report = validate_imports(
                roots=[
                    self.repo_root / "farfan_core" / "farfan_core" / "core",
                    self.repo_root / "farfan_core" / "farfan_core" / "entrypoint",
                    self.repo_root / "farfan_core" / "farfan_core" / "processing",
                ],
                import_policy=self.import_policy,
                repo_root=self.repo_root,
            )

            self.log_claim(
                "complete",
                "static_import_verification",
                                                 f"Static import analysis complete:
{len(static_report.static_import_violations)} violations",
                {"violation_count": len(static_report.static_import_violations)},
            )

            # Dynamic runtime verification (wraps rest of pipeline)
            dynamic_report = PolicyReport()

        except Exception as e:
            error_msg = f"Path/import verification setup failed: {e}"
            self.log_claim("error", "path_import_verification", error_msg)
            self.errors.append(error_msg)
            self.generate_verification_manifest([], {})
            return False

        # Wrap pipeline execution in path guard
        try:
            with guard_paths_and_imports(
                self.path_policy, self.import_policy, dynamic_report
            ):
                # Step 2: Run SPC ingestion (canonical phase-one)
                cpp = await self.run_spc_ingestion()
                if cpp is None:
                    self.path_import_report = merge_policy_reports(
                        [static_report, dynamic_report]
                    )
                    self.generate_verification_manifest([], {})
                    return False

                # Step 3: Run CPP adapter
                preprocessed_doc = await self.run_cpp_adapter(cpp)
                if preprocessed_doc is None:
                    self.path_import_report = merge_policy_reports(
                        [static_report, dynamic_report]
                    )
                    self.generate_verification_manifest([], {})
                    return False

                # Step 4: Run orchestrator
                results = await self.run_orchestrator(preprocessed_doc)
                if results is None:
                    self.path_import_report = merge_policy_reports(
                        [static_report, dynamic_report]
                    )
                    self.generate_verification_manifest([], {})
                    return False

        except Exception as e:
            error_msg = f"Pipeline execution failed under path guard: {e}"
            self.log_claim("error", "guarded_pipeline", error_msg)
            self.errors.append(error_msg)
            self.path_import_report = merge_policy_reports(
                [static_report, dynamic_report]
            )
            self.generate_verification_manifest([], {})
            return False

        # Merge static and dynamic reports
        self.path_import_report = merge_policy_reports([static_report, dynamic_report])

        self.log_claim(
            "complete",
            "path_import_verification",
                                               f"Path/import verification complete:
{self.path_import_report.violation_count()} total violations",
            {
                "static_violations": len(static_report.static_import_violations),
                "dynamic_violations": len(dynamic_report.dynamic_import_violations)
                + len(dynamic_report.path_violations),
                "success": self.path_import_report.ok(),
            },
        )

        # Step 5: Save artifacts
        artifacts, artifact_hashes = self.save_artifacts(cpp, preprocessed_doc, results)

        # Step 6: Generate verification manifest with chunk metrics
        manifest_path = self.generate_verification_manifest(
            artifacts, artifact_hashes, preprocessed_doc, results
        )

        self.log_claim(
            "complete",
            "pipeline",
            "Pipeline execution completed",
            {
                "success": self._last_manifest_success,
                "phases_completed": self.phases_completed,
                "phases_failed": self.phases_failed,
                "manifest_path": str(manifest_path),
            },
        )

        return bool(self._last_manifest_success)
```

```
def cli() -> None:
    """Synchronous entrypoint for console scripts."""
    try:
        # Perform module shadowing check before anything else
        # We do this here to catch it before main() potentially loads more things
        # Note: We duplicate the check logic here or rely on the one in global scope?
        # The global scope check raises RuntimeError. We need to catch that.
        # But the global scope code runs on import. So we can't catch it inside cli() if
we import this module.
        # Wait, this IS the module. When run as script, the global code runs.
        # To strictly comply, we should wrap the global check or move it.
        # Moving it to cli() is safer.

        # Check for module shadowing
        _expected_farfan_pipeline_prefix = (
            PROJECT_ROOT / "src" / "farfan_pipeline"
        ).resolve()
        if (
            not Path(farfan_pipeline.__file__)
            .resolve()
            .is_relative_to(_expected_farfan_pipeline_prefix)
        ):
            raise RuntimeError(
                "MODULE SHADOWING DETECTED!\n"
                f"  Expected farfan_pipeline from: {_expected_farfan_pipeline_prefix}\n"
                f"  Actually loaded from:  {farfan_pipeline.__file__}\n"
                "Fix: uninstall old package before running the verified pipeline."
            )

        asyncio.run(main())

    except RuntimeError as e:
        if "MODULE SHADOWING DETECTED" in str(e):
            print(f"\n? FATAL: {e}\n", flush=True)

            # Attempt to write minimal manifest
            try:
                # We need to guess artifacts dir since we haven't parsed args yet
                # Default is artifacts/plan1
                artifacts_dir = PROJECT_ROOT / "artifacts" / "plan1"
                artifacts_dir.mkdir(parents=True, exist_ok=True)

                manifest_path = artifacts_dir / "verification_manifest.json"
                manifest = {
                    "success": False,
                    "execution_id": datetime.utcnow().strftime("%Y%m%d_%H%M%S"),
                    "start_time": datetime.utcnow().isoformat(),
                    "end_time": datetime.utcnow().isoformat(),
                    "errors": [str(e)],
                    "artifacts_generated": [],
                    "artifact_hashes": {},
                    "phases_completed": 0,
                    "phases_failed": 1,
                }

                with open(manifest_path, "w") as f:
                    json.dump(manifest, f, indent=2)

                print(f"Manifest written to: {manifest_path}", flush=True)

            except Exception as manifest_err:
                print(f"Failed to write failure manifest: {manifest_err}", flush=True)

            print("PIPELINE_VERIFIED=0", flush=True)
            sys.exit(1)
        else:
            raise

    async def run_spc_ingestion(self) -> Optional[Any]:
        """
        Run SPC (Smart Policy Chunks) ingestion phase - canonical phase-one.

        Returns:
            SPC object if successful, None otherwise
        """
        self.log_claim("start", "spc_ingestion", "Starting SPC ingestion (phase-one)")

        try:
            from canonic_phases.Phase_one.phase0_input_validation import (
                Phase0Input,
                Phase0ValidationContract,
            )
            from canonic_phases.Phase_one.phase1_spc_ingestion_full import (
                execute_phase_1_with_full_contract,
            )

            # Phase 0: Validación
            phase0_input = Phase0Input(
                pdf_path=self.plan_pdf_path,
                run_id=self.execution_id,
                questionnaire_path=self.questionnaire_path,
            )
            phase0_contract = Phase0ValidationContract()
            canonical_input = await phase0_contract.execute(phase0_input)

            # Phase 1: Ingestion
            cpp = execute_phase_1_with_full_contract(canonical_input)

            self.phases_completed += 1
            self.log_claim(
                "complete",
                "spc_ingestion",
                "SPC ingestion (phase-one) completed successfully",
                {"phases_completed": self.phases_completed},
            )
            return cpp
        except Exception as e:
            self.phases_failed += 1
            error_msg = f"SPC ingestion failed: {str(e)}"
            self.log_claim(
                "error",
                "spc_ingestion",
                error_msg,
                {"traceback": traceback.format_exc()},
            )
            self.errors.append(error_msg)
            return None

    async def run_cpp_adapter(self, cpp: Any) -> Optional[Any]:
        """
        Run CPP adapter to convert CanonPolicyPackage to PreprocessedDocument.

        Args:
            cpp: CanonPolicyPackage from Phase 1 ingestion

        Returns:
            PreprocessedDocument if successful, None otherwise
        """
        self.log_claim("start", "cpp_adapter", "Starting CPP adaptation")

        try:
            from farfan_pipeline.utils.cpp_adapter import CPPAdapter

            # Derive document_id from CPP metadata or fallback to plan filename
            document_id = None
            if hasattr(cpp, "metadata") and isinstance(cpp.metadata, dict):
                document_id = cpp.metadata.get("document_id")
            if not document_id:
                document_id = self.plan_pdf_path.stem

            adapter = CPPAdapter()
            preprocessed = adapter.to_preprocessed_document(
                cpp, document_id=document_id
            )

            self.phases_completed += 1
            self.log_claim(
                "complete",
                "cpp_adapter",
                "CPP adaptation completed successfully",
                {
                    "phases_completed": self.phases_completed,
                    "document_id": document_id,
                },
            )
            return preprocessed

        except Exception as e:
            self.phases_failed += 1
            error_msg = f"CPP adaptation failed: {str(e)}"
            self.log_claim(
                "error",
                "cpp_adapter",
                error_msg,
                {"traceback": traceback.format_exc()},
            )
            self.errors.append(error_msg)
            return None

    async def run_orchestrator(self, preprocessed_doc: Any) -> Optional[list[Any]]:
        """
        Run orchestrator with all phases and verify Phase 2 success.

        Args:
            preprocessed_doc: PreprocessedDocument

        Returns:
            List of PhaseResult objects if successful, None otherwise
        """
        self.log_claim("start", "orchestrator", "Starting orchestrator execution")

        try:
            # This is not the PhaseOrchestrator from the other file, but the core one.
            from orchestration.factory import build_processor

            processor = build_processor()

            # The core orchestrator is at processor.orchestrator
            results = await processor.orchestrator.process_development_plan_async(
                pdf_path=str(self.plan_pdf_path), preprocessed_document=preprocessed_doc
            )

            # Capture Phase 2 metrics directly from orchestrator
            if hasattr(processor.orchestrator, "_execution_metrics"):
                self.phase2_metrics = processor.orchestrator._execution_metrics.get(
                    "phase_2"
                )

            if not results:
                raise RuntimeError("Orchestrator returned no results.")

            # JOBFRONT 3: Verify Phase 2 (Microquestions) success
            phase2_ok = False
            phase2_report = {"success": False, "question_count": 0, "errors": []}
            if len(results) >= 3:
                phase2_result = results[2]  # This is a PhaseResult dataclass
                if phase2_result.success:
                    is_valid, validation_errors, normalized_questions = (
                        validate_phase2_result(phase2_result.data)
                    )
                    if is_valid:
                        phase2_ok = True
                        phase2_report["success"] = True
                        phase2_report["question_count"] = len(
                            normalized_questions or []
                        )
                    else:
                          error_msg = "Orchestrator Phase 2 failed structural invariant:
questions list is empty or missing."
                        phase2_report["errors"].extend(validation_errors or [])
                        phase2_report["errors"].append(error_msg)
                        self.log_claim(
                            "error",
                            "orchestrator",
                            error_msg,
                            {"phase_id": phase2_result.phase_id},
                        )
                        self.errors.append(error_msg)
                else:
                    error_msg = (
                        f"Orchestrator Phase 2 failed internally: {phase2_result.error}"
                    )
                    phase2_report["errors"].append(error_msg)
                    self.log_claim(
                        "error",
                        "orchestrator",
                        error_msg,
                        {"phase_id": phase2_result.phase_id},
                    )
                    self.errors.append(error_msg)
            else:
                error_msg = "Orchestrator did not produce a result for Phase 2."
                phase2_report["errors"].append(error_msg)
                self.log_claim("error", "orchestrator", error_msg)
                self.errors.append(error_msg)

            self.phase2_report = phase2_report

            if not phase2_ok:
                # Signal failure as per this script's convention
                self.phases_failed += 1
                return None

            # Correctly count completed phases from the results list
            completed_phases = sum(1 for r in results if r.success)
            self.phases_completed += completed_phases

            self.log_claim(
                "complete",
                "orchestrator",
                "Orchestrator execution completed successfully",
                {
                    "phases_completed": self.phases_completed,
                    "core_phases_run": len(results),
                },
            )
            return results
```

```
        except Exception as e:
            self.phases_failed += 1
            error_msg = f"Orchestrator execution failed: {str(e)}"
            self.log_claim(
                "error",
                "orchestrator",
                error_msg,
                {"traceback": traceback.format_exc()},
            )
            self.errors.append(error_msg)
            if self.phase2_report is None:
                self.phase2_report = {
                    "success": False,
                    "question_count": 0,
                    "errors": [error_msg],
                }
            return None

    def save_artifacts(
        self, cpp: Any, preprocessed_doc: Any, results: Any
    ) -> tuple[List[str], Dict[str, str]]:
        """
        Save artifacts and compute hashes.

        Args:
            cpp: CPP object
            preprocessed_doc: PreprocessedDocument
            results: Orchestrator results

        Returns:
            List of artifact file paths
        """
        self.log_claim("start", "artifact_generation", "Saving artifacts")

        artifacts = []
        artifact_hashes = {}

        try:
            # Save complete CanonPolicyPackage if available (HOSTILE AUDIT REQUIREMENT)
            if cpp:
                cpp_path = self.artifacts_dir / "cpp.json"
                try:
                    # Serialize CPP with custom JSON encoder for dataclasses
                    from dataclasses import asdict, is_dataclass
                    import numpy as np

                    def cpp_to_dict(obj):
                        """Convert dataclass/numpy to JSON-serializable format"""
                        if is_dataclass(obj):
                            return asdict(obj)
                        elif isinstance(obj, np.ndarray):
                            return obj.tolist()
                        elif isinstance(obj, (np.int64, np.int32)):
```

```
                        return int(obj)
                    elif isinstance(obj, (np.float64, np.float32)):
                        return float(obj)
                    else:
                        return str(obj)

                cpp_dict = asdict(cpp) if is_dataclass(cpp) else {}

                with open(cpp_path, "w") as f:
                    json.dump(cpp_dict, f, indent=2, default=cpp_to_dict)

                artifacts.append(str(cpp_path))
                artifact_hashes[str(cpp_path)] = self.compute_sha256(cpp_path)

                self.log_claim(
                    "artifact",
                    "cpp_serialization",
                    f"Serialized complete CanonPolicyPackage",
                    {"file": str(cpp_path), "size_bytes": cpp_path.stat().st_size},
                )

            except Exception as e:
                self.log_claim(
                    "error",
                    "artifact_generation",
                    f"Failed to serialize CPP: {str(e)}",
                )

        # Save preprocessed document metadata
        if preprocessed_doc:
            doc_metadata_path = (
                self.artifacts_dir / "preprocessed_doc_metadata.json"
            )
            try:
                with open(doc_metadata_path, "w") as f:
                    json.dump(
                        {
                            "execution_id": self.execution_id,
                            "doc_generated": True,
                            "timestamp": datetime.utcnow().isoformat(),
                        },
                        f,
                        indent=2,
                    )
                artifacts.append(str(doc_metadata_path))
                artifact_hashes[str(doc_metadata_path)] = self.compute_sha256(
                    doc_metadata_path
                )
            except Exception as e:
                self.log_claim(
                    "error",
                    "artifact_generation",
                    f"Failed to save doc metadata: {str(e)}",
                )
        # Save results summary
        if results:
            results_path = self.artifacts_dir / "results_summary.json"
            try:
                with open(results_path, "w") as f:
                    json.dump(
                        {
                            "execution_id": self.execution_id,
                            "results_generated": True,
                            "timestamp": datetime.utcnow().isoformat(),
                        },
                        f,
                        indent=2,
                    )
                artifacts.append(str(results_path))
                artifact_hashes[str(results_path)] = self.compute_sha256(
                    results_path
                )
            except Exception as e:
                self.log_claim(
                    "error",
                    "artifact_generation",
                    f"Failed to save results: {str(e)}",
                )

        # Save all claims
        claims_path = self.artifacts_dir / "execution_claims.json"
        with open(claims_path, "w") as f:
            json.dump([claim.to_dict() for claim in self.claims], f, indent=2)
        artifacts.append(str(claims_path))
        artifact_hashes[str(claims_path)] = self.compute_sha256(claims_path)

        self.log_claim(
            "complete",
            "artifact_generation",
            f"Saved {len(artifacts)} artifacts",
            {"artifact_count": len(artifacts)},
        )

        return artifacts, artifact_hashes

    except Exception as e:
        error_msg = f"Failed to save artifacts: {str(e)}"
        self.log_claim("error", "artifact_generation", error_msg)
        self.errors.append(error_msg)
        return artifacts, artifact_hashes

def _collect_calibration_manifest_data(self) -> Dict[str, Any]:
    """Collect calibration metadata for manifest inclusion."""
    calibration_file = PROJECT_ROOT / "config" / "intrinsic_calibration.json"
    if not calibration_file.exists():
        return {}
    try:
        with open(calibration_file, encoding="utf-8") as handle:
            calibration_payload = json.load(handle)

        calibration_hash = hashlib.sha256(
            json.dumps(calibration_payload, sort_keys=True).encode("utf-8")
        ).hexdigest()

        return {
            "version": self.versions.get("calibration"),
            "hash": calibration_hash[:16],
            "methods_calibrated": len(calibration_payload),
            "methods_missing": [],
        }
    except Exception as exc:
        self.log_claim(
            "warning",
            "calibration_manifest",
            f"Unable to read calibration data: {exc}",
            {"path": str(calibration_file)},
        )
        return {}

def _calculate_chunk_metrics(
    self,
    preprocessed_doc: Any,
    results: Any,
    phase2_metrics: Dict[str, Any] | None = None,
) -> Dict[str, Any]:
    """
    Calculate SPC utilization metrics for verification manifest.

    Args:
        preprocessed_doc: PreprocessedDocument with chunk information
        results: Orchestrator execution results
        phase2_metrics: Optional metrics dictionary from orchestrator

    Returns:
        Dictionary with chunk metrics
    """
    if preprocessed_doc is None:
        return {}

    processing_mode = getattr(preprocessed_doc, "processing_mode", "flat")

    if processing_mode != "chunked":
        return {
            "processing_mode": "flat",
            "note": "Document processed in flat mode (no chunk utilization)",
        }

    chunks = getattr(preprocessed_doc, "chunks", [])
    chunk_graph = getattr(preprocessed_doc, "chunk_graph", {})
```

```python
chunk_metrics = {
    "processing_mode": "chunked",
    "total_chunks": len(chunks),
    "chunk_types": {},
    "chunk_routing": {},
    "graph_metrics": {},
    "execution_savings": {},
    "provenance_coverage": 0.0,
}

# Count chunk types and provenance
chunks_with_provenance = 0
for chunk in chunks:
    chunk_type = getattr(chunk, "chunk_type", "unknown")
    chunk_metrics["chunk_types"][chunk_type] = (
        chunk_metrics["chunk_types"].get(chunk_type, 0) + 1
    )

    # Check provenance
    if hasattr(chunk, "provenance") and chunk.provenance:
        # Strict check: must have page_number
        if getattr(chunk.provenance, "page_number", None) is not None:
            chunks_with_provenance += 1

if len(chunks) > 0:
    chunk_metrics["provenance_coverage"] = round(
        chunks_with_provenance / len(chunks), 4
    )

# Calculate graph metrics if networkx available
try:
    import networkx as nx

    if chunk_graph and isinstance(chunk_graph, dict):
        nodes = chunk_graph.get("nodes", [])
        edges = chunk_graph.get("edges", [])

        # Build networkx graph for analysis
        G = nx.DiGraph()
        for node in nodes:
            node_id = node.get("id")
            if node_id is not None:
                G.add_node(node_id)

        for edge in edges:
            source = edge.get("source")
            target = edge.get("target")
            if source is not None and target is not None:
                G.add_edge(source, target)

        chunk_metrics["graph_metrics"] = {
            "nodes": G.number_of_nodes(),
            "edges": G.number_of_edges(),
            "is_dag": nx.is_directed_acyclic_graph(G),
            "is_connected": (
                nx.is_weakly_connected(G) if G.number_of_nodes() > 0 else False
            ),
            "density": (
                round(nx.density(G), 4) if G.number_of_nodes() > 0 else 0.0
            ),
        }

        # Calculate diameter if connected
        if chunk_metrics["graph_metrics"]["is_connected"]:
            try:
                chunk_metrics["graph_metrics"]["diameter"] = nx.diameter(
                    G.to_undirected()
                )
            except Exception:
                chunk_metrics["graph_metrics"]["diameter"] = -1
        else:
            chunk_metrics["graph_metrics"]["diameter"] = -1

except ImportError:
    chunk_metrics["graph_metrics"] = {
        "note": "NetworkX not available for graph analysis"
    }
except Exception as e:
    chunk_metrics["graph_metrics"] = {
        "error": f"Graph analysis failed: {str(e)}"
    }

# Calculate execution savings
# Use actual metrics from orchestrator if available
if phase2_metrics:
    metrics = phase2_metrics
    chunk_metrics["execution_savings"] = {
        "chunk_executions": metrics.get("chunk_executions", 0),
        "full_doc_executions": metrics.get("full_doc_executions", 0),
        "total_possible_executions": metrics.get(
            "total_possible_executions", 0
        ),
        "actual_executions": metrics.get("actual_executions", 0),
        "savings_percent": round(metrics.get("savings_percent", 0.0), 2),
        "routing_table_version": metrics.get(
            "routing_table_version", "unknown"
        ),
        "note": "Actual execution counts from orchestrator Phase 2",
    }
elif results:
    # Fallback to estimation if real metrics not available
    total_possible_executions = 30 * len(chunks)  # 30 executors per chunk max
    # Assume chunk routing reduces executions by using type-specific executors
    estimated_actual = (
        len(chunks) * 10
    )  # ~10 executors per chunk (conservative)

    chunk_metrics["execution_savings"] = {
            "total_possible_executions": total_possible_executions,
            "estimated_actual_executions": estimated_actual,
            "estimated_savings_percent": (
                round(
                    (1 - estimated_actual / max(total_possible_executions, 1))
                    * 100,
                    2,
                )
                if total_possible_executions > 0
                else 0.0
            ),
              "note": "Estimated savings based on chunk-aware routing (orchestrator
metrics not available)",
        }

    return chunk_metrics

def _calculate_signal_metrics(self, results: Any) -> Dict[str, Any]:
    """
    Calculate signal utilization metrics for verification manifest.

    Args:
        results: Orchestrator execution results

    Returns:
        Dictionary with signal metrics
    """
    # Try to extract signal usage from results
    try:
        signal_metrics = {
            "enabled": True,
            "transport": "memory",
            "policy_areas_loaded": 10,
        }

        # Check if results have executor information
        if results and hasattr(results, "executor_metadata"):
            # Count executors that used signals
            executors_with_signals = 0
            total_executors = 0

            for metadata in results.executor_metadata.values():
                total_executors += 1
                if metadata.get("signal_usage"):
                    executors_with_signals += 1

            signal_metrics["executors_using_signals"] = executors_with_signals
            signal_metrics["total_executors"] = total_executors

        # Default values if we can't extract from results
        if "executors_using_signals" not in signal_metrics:
            signal_metrics["executors_using_signals"] = 0
            signal_metrics["total_executors"] = 0
            signal_metrics["note"] = (
                    "Signal infrastructure initialized, actual usage not tracked in results"
            )

        # Add signal pack versions
        signal_metrics["signal_versions"] = {
            f"PA{i:02d}": "1.0.0" for i in range(1, 11)
        }

        return signal_metrics

    except Exception as e:
        # If signal system not initialized, return minimal info
        return {
            "enabled": False,
            "note": f"Signal system not initialized: {str(e)}",
        }

def _extract_synchronization_data(self, results: Any) -> Dict[str, Any]:
    """
    Extract synchronization plan data from orchestrator results.

    Args:
        results: Orchestrator execution results (list of PhaseResult objects)

    Returns:
        Dictionary with synchronization plan metadata
    """
    try:
        synchronization_data = {
            "plan_id": None,
            "integrity_hash": None,
            "task_count": 0,
            "chunk_count": 0,
            "question_count": 0,
            "correlation_id": None,
            "created_at": None,
        }

        if not results:
            return synchronization_data

        for result in results:
            if hasattr(result, "data") and isinstance(result.data, dict):
                if "_execution_plan" in result.data:
                    plan = result.data["_execution_plan"]
                    if hasattr(plan, "plan_id"):
                        synchronization_data["plan_id"] = plan.plan_id
                        synchronization_data["integrity_hash"] = plan.integrity_hash
                        synchronization_data["task_count"] = len(plan.tasks)
                        synchronization_data["chunk_count"] = plan.chunk_count
                        synchronization_data["question_count"] = plan.question_count
                        synchronization_data["correlation_id"] = plan.correlation_id
                        synchronization_data["created_at"] = plan.created_at
                        return synchronization_data

        return None

    except Exception as e:
        self.log_claim(
            "warning",
            "synchronization_extraction",
            f"Unable to extract synchronization data: {e}",
        )
        return None

def generate_verification_manifest(
    self,
    artifacts: List[str],
    artifact_hashes: Dict[str, str],
    preprocessed_doc: Any = None,
    results: Any = None,
) -> Path:
    """
          Generate final verification manifest with SPC utilization metrics and cryptographic integrity.

    Args:
        artifacts: List of artifact paths
        artifact_hashes: Dictionary mapping paths to SHA256 hashes
        preprocessed_doc: PreprocessedDocument (optional, for chunk metrics)
        results: Orchestrator results (optional, for execution metrics)

    Returns:
        Path to verification_manifest.json
    """
    end_time = datetime.utcnow().isoformat()

    # Calculate chunk utilization metrics
    chunk_metrics = self._calculate_chunk_metrics(
        preprocessed_doc, results, getattr(self, "phase2_metrics", None)
    )

    # HOSTILE AUDIT: Validate critical invariants before declaring success
    hostile_failures: list[str] = []

    if preprocessed_doc:
        chunk_count = len(getattr(preprocessed_doc, "chunks", []))
        if chunk_count < 5:
            hostile_failures.append(f"chunk_graph too small: {chunk_count} < 5")

        # === PHASE 2 HARDENING: STRICT SPC INVARIANTS ===
        # Enforce exactly 60 chunks and chunked mode for SPC ingestion
        if chunk_metrics.get("processing_mode") != "chunked":
            hostile_failures.append(
                f"Invalid processing_mode: {chunk_metrics.get('processing_mode')} != chunked"
            )
        if chunk_metrics.get("total_chunks") != 60:
            hostile_failures.append(
                f"Invalid total_chunks: {chunk_metrics.get('total_chunks')} != 60"
            )

        # Enforce Provenance Coverage using Calibrated Threshold
        # SOTA: No hardcoded values. Use centralized calibration.
         #               from farfan_core import get_parameter_loader  # CALIBRATION
DISABLED
        #             param_loader = get_parameter_loader()  # CALIBRATION DISABLED

        # Fetch threshold for this specific method
                                                                        method_key      =
"farfan_core.scripts.run_policy_pipeline_verified.VerifiedPipelineRunner.generate_verifi
cation_manifest"
             #                 calibrated_params = param_loader.get(method_key)   #
CALIBRATION DISABLED

        # Default to 1.0 (strict) if not found, but log warning if falling back
        required_coverage = calibrated_params.get(
            "provenance_coverage_threshold", 1.0
        )

        provenance_coverage = chunk_metrics.get("provenance_coverage", 0.0)
        if provenance_coverage < required_coverage:
            hostile_failures.append(
                           f"Provenance  coverage  violation:  {provenance_coverage}  <
{required_coverage} (Threshold from {method_key})"
            )

    phase2_entry = {
        "name": "Phase 2 ? Micro Questions",
        "success": bool(self.phase2_report and self.phase2_report.get("success")),
        "question_count": (self.phase2_report or {}).get("question_count", 0),
        "errors": list((self.phase2_report or {}).get("errors", [])),
    }
    if not phase2_entry["success"] and not phase2_entry["errors"]:
        phase2_entry["errors"].append("Phase 2 not executed")

    # Determine success based on strict criteria + hostile invariants
    # We start assuming success is possible, then disqualify based on failures
    success = True

    if self._bootstrap_failed:
        success = False
    if self.phases_failed > 0:
        success = False
    if self.phases_completed == 0:
        success = False
    if len(self.errors) > 0:
        success = False
    if len(artifacts) == 0:
        success = False
```

```
if len(hostile_failures) > 0:
    success = False
if not phase2_entry["success"]:
    success = False
if self.path_import_report and not self.path_import_report.ok():
    success = False

if hostile_failures:
    self.log_claim(
        "error",
        "hostile_audit",
        f"Hostile audit failures: {hostile_failures}",
    )
    self.errors.extend(hostile_failures)

builder = self.manifest_builder
builder.manifest_data["versions"] = dict(self.versions)

# Set environment with strict error handling
try:
    builder.set_environment()
except Exception as e:
    error_msg = f"Failed to set environment in manifest: {e}"
    self.log_claim("error", "environment", error_msg)
    self.errors.append(error_msg)
    success = False

# Set pipeline hash with strict validation
pipeline_hash = getattr(self, "input_pdf_sha256", "")
if not pipeline_hash:
    error_msg = "Missing input PDF hash for manifest"
    self.log_claim("error", "input_verification", error_msg)
    self.errors.append(error_msg)
    success = False

builder.set_pipeline_hash(pipeline_hash)

# Set path/import verification results
if self.path_import_report:
    builder.set_path_import_verification(self.path_import_report)

# Update success status in builder and self
self._last_manifest_success = success
builder.set_success(success)

# Determinism metadata
seed_entry = self.seed_registry.get_manifest_entry(
    policy_unit_id=self.policy_unit_id,
    correlation_id=self.correlation_id,
)
builder.set_determinism(
    seed_version=seed_entry.get("seed_version", ""),
    policy_unit_id=seed_entry.get("policy_unit_id"),
    correlation_id=seed_entry.get("correlation_id"),
    seeds_by_component=seed_entry.get("seeds_by_component"),
)

# Calibration metadata
calibration_manifest = self._collect_calibration_manifest_data()
if calibration_manifest:
    builder.set_calibrations(
        calibration_manifest["version"],
        calibration_manifest["hash"],
        calibration_manifest["methods_calibrated"],
        calibration_manifest["methods_missing"],
    )

# Ingestion metadata
if preprocessed_doc:
    raw_text = getattr(preprocessed_doc, "raw_text", "") or ""
    sentences = getattr(preprocessed_doc, "sentences", []) or []
    chunk_count = len(getattr(preprocessed_doc, "chunks", []))
    builder.set_ingestion(
        method="SPC",
        chunk_count=chunk_count,
        text_length=len(raw_text),
        sentence_count=len(sentences),
        chunk_strategy="semantic",
        chunk_overlap=50,
    )

builder.manifest_data.setdefault("phases", {})
builder.manifest_data["phases"]["phase2"] = phase2_entry

# Phase metadata
duration_seconds = (
    datetime.fromisoformat(end_time) - datetime.fromisoformat(self.start_time)
).total_seconds()
builder.add_phase(
    phase_id=0,
    phase_name="complete_pipeline",
    success=success,
    duration_ms=int(duration_seconds * 1000),
    items_processed=self.phases_completed,
    error="; ".join(self.errors) if self.errors and not success else None,
)

# Artifacts
for index, artifact_path in enumerate(sorted(artifact_hashes.keys())):
    artifact_file = Path(artifact_path)
    size_bytes = (
        artifact_file.stat().st_size if artifact_file.exists() else None
    )
    builder.add_artifact(
        artifact_id=f"artifact_{index:02d}",
        path=str(artifact_file),
        artifact_hash=artifact_hashes[artifact_path],
        size_bytes=size_bytes,
    )

if hasattr(self, "questionnaire_sha256"):
    questionnaire_size = (
        self.questionnaire_path.stat().st_size
        if self.questionnaire_path.exists()
        else None
    )
    builder.add_artifact(
        artifact_id="questionnaire_source",
        path=str(self.questionnaire_path),
        artifact_hash=self.questionnaire_sha256,
        size_bytes=questionnaire_size,
    )
    self.log_claim(
        "artifact",
        "questionnaire",
        "Questionnaire added to manifest",
        {
            "path": str(self.questionnaire_path),
            "hash": self.questionnaire_sha256,
        },
    )

if chunk_metrics:
    builder.set_spc_utilization(chunk_metrics)

signal_metrics = self._calculate_signal_metrics(results)
if signal_metrics:
    builder.manifest_data["signals"] = signal_metrics

synchronization_data = self._extract_synchronization_data(results)
if synchronization_data:
    builder.manifest_data["synchronization"] = synchronization_data

builder.manifest_data.update(
    {
        "execution_id": self.execution_id,
        "start_time": self.start_time,
        "end_time": end_time,
        "input_pdf_path": str(self.plan_pdf_path),
        "total_claims": len(self.claims),
        "errors": list(self.errors),
        "artifacts_generated": list(artifacts),
        "artifact_hashes": dict(artifact_hashes),
    }
)

manifest_path = self.artifacts_dir / "verification_manifest.json"
manifest_dict = builder.build()
manifest_path.write_text(json.dumps(manifest_dict, indent=2), encoding="utf-8")

hmac_secret = builder.hmac_secret
is_valid = True
```

```
        if hmac_secret:
            is_valid = verify_manifest_integrity(manifest_dict, hmac_secret)
            if is_valid:
                self.log_claim(
                    "hash",
                    "verification_manifest",
                    "Manifest integrity verified",
                    {"file": str(manifest_path)},
                )
            else:
                self.log_claim(
                    "error",
                    "verification_manifest",
                    "Manifest integrity verification failed",
                )
        else:
            self.log_claim(
                "warning",
                "verification_manifest",
                "No HMAC secret provided; integrity verification skipped",
            )

        if success and is_valid:
            print("\n" + "=" * 80)
            print("PIPELINE_VERIFIED=1")
            print(f"Manifest: {manifest_path}")
            print(f"HMAC: {manifest_dict.get('integrity_hmac', 'N/A')[:16]}...")
            print(
                f"Phases: {self.phases_completed} completed, {self.phases_failed}
failed"
            )
            print(f"Artifacts: {len(artifacts)}")
            print("=" * 80 + "\n")

        return manifest_path

    async def run(self) -> bool:
        """
        Execute the complete verified pipeline.

        Returns:
            True if pipeline succeeded, False otherwise
        """
        # Check for bootstrap failures (Phase 0.0)
        if self._bootstrap_failed:
            self.generate_verification_manifest([], {})
            return False

        self.log_claim("start", "pipeline", "Starting verified pipeline execution")

        # Step 1: Verify input
        if not self.verify_input():
            self.generate_verification_manifest([], {})
            return False
        # Step 1.5: Run boot checks
        try:
             # Ensure runtime_config is available (should be if bootstrap passed, but be safe)
            if self.runtime_config is None:
                raise BootCheckError(
                    "Runtime config is None",
                    "BOOT_CONFIG_MISSING",
                    "Runtime config not initialized",
                )

            if not self.run_boot_checks():
                # Boot checks failed but we're in DEV mode - log warning
                self.log_claim(
                    "warning",
                    "boot_checks",
                    "Boot checks failed but continuing in non-PROD mode",
                )
        except BootCheckError:
            # Boot check failed in PROD mode - abort
            self.generate_verification_manifest([], {})
            return False

        # Step 2: Run SPC ingestion (canonical phase-one)
        cpp = await self.run_spc_ingestion()
        if cpp is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 3: Run CPP adapter
        preprocessed_doc = await self.run_cpp_adapter(cpp)
        if preprocessed_doc is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 4: Run orchestrator
        results = await self.run_orchestrator(preprocessed_doc)
        if results is None:
            self.generate_verification_manifest([], {})
            return False

        # Step 5: Save artifacts
        artifacts, artifact_hashes = self.save_artifacts(cpp, preprocessed_doc, results)

        # Step 6: Generate verification manifest with chunk metrics
        manifest_path = self.generate_verification_manifest(
            artifacts, artifact_hashes, preprocessed_doc, results
        )

        self.log_claim(
            "complete",
            "pipeline",
            "Pipeline execution completed",
            {
                "success": self._last_manifest_success,
                "phases_completed": self.phases_completed,
                "phases_failed": self.phases_failed,
                "manifest_path": str(manifest_path),
            },
        )

        return bool(self._last_manifest_success)


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

    parser = argparse.ArgumentParser(
        description="Run verified policy pipeline with cryptographic verification"
    )
    parser.add_argument(
        "--plan",
        type=str,
        default="data/plans/Plan_1.pdf",
        help="Path to plan PDF (default: data/plans/Plan_1.pdf)",
    )
    parser.add_argument(
        "--artifacts-dir",
        type=str,
        default="artifacts/plan1",
        help="Directory for artifacts (default: artifacts/plan1)",
    )

    args = parser.parse_args()

    # Resolve paths
    plan_path = PROJECT_ROOT / args.plan
    artifacts_dir = PROJECT_ROOT / args.artifacts_dir

    print("=" * 80, flush=True)
    print("F.A.R.F.A.N VERIFIED POLICY PIPELINE RUNNER", flush=True)
    print("Framework for Advanced Retrieval of Administrativa Narratives", flush=True)
    print("=" * 80, flush=True)
    print(f"Plan: {plan_path}", flush=True)
    print(f"Artifacts: {artifacts_dir}", flush=True)
    print("=" * 80, flush=True)

    # Create and run pipeline
    runner = VerifiedPipelineRunner(plan_path, artifacts_dir)
    success = await runner.run()

    print("=" * 80, flush=True)
    if success:
        print("PIPELINE_VERIFIED=1", flush=True)
        print("Status: SUCCESS", flush=True)
    else:
        print("PIPELINE_VERIFIED=0", flush=True)
        print("Status: FAILED", flush=True)
    print("=" * 80, flush=True)

    sys.exit(0 if success else 1)


def cli() -> None:
    """Synchronous entrypoint for console scripts."""
    try:
        # Check for module shadowing before anything else
        _expected_farfan_pipeline_prefix = (
            PROJECT_ROOT / "src" / "farfan_pipeline"
        ).resolve()
        if (
            not Path(farfan_pipeline.__file__)
            .resolve()
            .is_relative_to(_expected_farfan_pipeline_prefix)
        ):
            raise RuntimeError(
                "MODULE SHADOWING DETECTED!\n"
                f"  Expected farfan_pipeline from: {_expected_farfan_pipeline_prefix}\n"
                f"  Actually loaded from:  {farfan_pipeline.__file__}\n"
                "Fix: uninstall old package before running the verified pipeline."
            )

        asyncio.run(main())

    except RuntimeError as e:
        if "MODULE SHADOWING DETECTED" in str(e):
            print(f"\n? FATAL: {e}\n", flush=True)

            # Attempt to write minimal manifest
            try:
                # We need to guess artifacts dir since we haven't parsed args yet
                # Default is artifacts/plan1
                artifacts_dir = PROJECT_ROOT / "artifacts" / "plan1"
                artifacts_dir.mkdir(parents=True, exist_ok=True)

                manifest_path = artifacts_dir / "verification_manifest.json"
                manifest = {
                    "success": False,
                    "execution_id": datetime.utcnow().strftime("%Y%m%d_%H%M%S"),
                    "start_time": datetime.utcnow().isoformat(),
                    "end_time": datetime.utcnow().isoformat(),
                    "errors": [str(e)],
                    "artifacts_generated": [],
                    "artifact_hashes": {},
                    "phases_completed": 0,
                    "phases_failed": 1,
                }

                with open(manifest_path, "w") as f:
                    json.dump(manifest, f, indent=2)
                print(f"Manifest written to: {manifest_path}", flush=True)

            except Exception as manifest_err:
                print(f"Failed to write failure manifest: {manifest_err}", flush=True)

            print("PIPELINE_VERIFIED=0", flush=True)
            sys.exit(1)
        else:
            raise


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

src/farfan_pipeline/phases/Phase_zero/paths.py

```python
"""
Portable, secure, and deterministic path utilities for SAAAAAA.

This module provides cross-platform path operations that ensure:
- Portability across Linux, macOS, and Windows
- Security through path traversal protection
- Determinism via normalized paths
- Controlled write locations (never in source tree)

All path operations in the repository MUST use these utilities instead of:
- Direct __file__ usage for resource access
- sys.path manipulation
- Hardcoded absolute paths
- os.path functions (use pathlib.Path instead)
"""

from __future__ import annotations

import os
import unicodedata
from pathlib import Path
from typing import Final


# Custom exception types for path errors
class PathError(Exception):
    """Base exception for path-related errors."""
    pass


class PathTraversalError(PathError):
    """Raised when a path attempts to escape workspace boundaries."""
    pass


class PathNotFoundError(PathError):
    """Raised when a required path does not exist."""
    pass


class PathOutsideWorkspaceError(PathError):
    """Raised when a path is outside the allowed workspace."""
    pass


class UnnormalizedPathError(PathError):
    """Raised when a path is not properly normalized."""
    pass


# Project root detection - computed once at module load
def _detect_project_root() -> Path:
    """
    Detect the project root directory using filesystem markers.

    This function uses a multi-strategy approach to locate the project root:

    1. Primary strategy: Search for pyproject.toml
       - Walks up the directory tree from this file's location
       - Returns the first directory containing pyproject.toml

    2. Secondary strategy: Search for src/farfan_pipeline (or legacy src/farfan_core) layout
       - Looks for directories with src/farfan_pipeline (or src/farfan_core) and setup.py
       - This supports older project structures

    3. Fallback strategy: Relative path calculation
       - If no markers found, assumes standard layout (src/<package>/utils)
       - Returns path 3 levels up from this file

    The function is called once at module load time, and the result is
    cached in the PROJECT_ROOT constant.

    Returns:
        Path: Absolute path to the project root directory

    Raises:
        No exceptions raised; always returns a path (uses fallback if needed)

    Note:
        This function is intended for internal use. External code should use
        the PROJECT_ROOT constant instead of calling this directly.
    """
    # Start from this file's location
    current = Path(__file__).resolve().parent

    # Walk up to find pyproject.toml
    for parent in [current] + list(current.parents):
        if (parent / "pyproject.toml").exists():
            return parent
        if (
            ((parent / "src" / "farfan_pipeline").exists() or (parent / "src" / "farfan_core").exists())
            and (parent / "setup.py").exists()
        ):
            return parent

    # Fallback: if we can't find it, assume we're in src/<package>/utils
    # and go up 3 levels
    return current.parent.parent.parent


# Global constants for common directories
PROJECT_ROOT: Final[Path] = _detect_project_root()
SRC_DIR: Final[Path] = PROJECT_ROOT / "src"
```

```
DATA_DIR: Final[Path] = PROJECT_ROOT / "data"
TESTS_DIR: Final[Path] = PROJECT_ROOT / "tests"
CONFIG_DIR: Final[Path] = PROJECT_ROOT / "canonic_questionnaire_central"
QUESTIONNAIRE_FILE: Final[Path] = CONFIG_DIR / "questionnaire_monolith.json"


def proj_root() -> Path:
    """
    Get the project root directory.

    Returns:
        Absolute path to the project root (where pyproject.toml lives)
    """
    return PROJECT_ROOT


def src_dir() -> Path:
    """Get the src directory path."""
    return SRC_DIR


def data_dir() -> Path:
    """
    Get the data directory path.
    Creates it if it doesn't exist.
    """
    DATA_DIR.mkdir(parents=True, exist_ok=True)
    return DATA_DIR


def tmp_dir() -> Path:
    """
    Get a project-specific temporary directory.

    Uses PROJECT_ROOT/tmp to keep temporary files within the workspace
    and avoid polluting system temp directories.

    Returns:
        Path to tmp directory (created if needed)
    """
    tmp = PROJECT_ROOT / "tmp"
    tmp.mkdir(parents=True, exist_ok=True)
    return tmp


def build_dir() -> Path:
    """
    Get the build directory for generated artifacts.

    Returns:
        Path to build directory (created if needed)
    """
    build = PROJECT_ROOT / "build"
    build.mkdir(parents=True, exist_ok=True)
    return build


def cache_dir() -> Path:
    """
    Get the cache directory.

    Returns:
        Path to cache directory (created if needed)
    """
    cache = build_dir() / "cache"
    cache.mkdir(parents=True, exist_ok=True)
    return cache


def reports_dir() -> Path:
    """
    Get the reports directory for generated reports.

    Returns:
        Path to reports directory (created if needed)
    """
    reports = build_dir() / "reports"
    reports.mkdir(parents=True, exist_ok=True)
    return reports


def is_within(base: Path, child: Path) -> bool:
    """
    Check if child path is within base directory (no traversal outside).

    Args:
        base: Base directory that should contain child
        child: Path to check

    Returns:
        True if child is within base, False otherwise

    Example:
        >>> project_root = Path("project_root")
        >>> is_within(project_root, project_root / "src" / "file.py")
        True
        >>> other_root = Path("other_project")
        >>> is_within(project_root, other_root / "file.py")
        False
    """
    try:
        base_resolved = base.resolve()
        child_resolved = child.resolve()

        # Check if child is relative to base
        child_resolved.relative_to(base_resolved)
        return True
    except (ValueError, RuntimeError):
        return False


def safe_join(base: Path, *parts: str) -> Path:
    """
    Safely join path components, preventing traversal outside base.

    This prevents directory traversal attacks using ".." components.

    Args:
        base: Base directory
        *parts: Path components to join

    Returns:
        Resolved path within base

    Raises:
        PathTraversalError: If the resulting path would be outside base

    Example:
        >>> project_root = Path("project_root")
        >>> safe_join(project_root, "src", "file.py")
        project_root/src/file.py
        >>> safe_join(project_root, "..", "other")  # raises
        PathTraversalError
    """
    result = base.joinpath(*parts).resolve()

    if not is_within(base, result):
        raise PathTraversalError(
            f"Path traversal detected: '{result}' is outside base '{base}'. "
            f"Use paths within the workspace."
        )

    return result


def normalize_unicode(path: Path, form: str = "NFC") -> Path:
    """
    Normalize Unicode in path for cross-platform consistency.

    Different filesystems handle Unicode differently:
    - macOS (HFS+) uses NFD normalization
    - Linux typically uses NFC
    - Windows uses UTF-16

    Args:
        path: Path to normalize
        form: Unicode normalization form ("NFC", "NFD", "NFKC", "NFKD")
              Default "NFC" for maximum compatibility

    Returns:
        Path with normalized Unicode
    """
    normalized_str = unicodedata.normalize(form, str(path))
    return Path(normalized_str)


def normalize_case(path: Path) -> Path:
    """
    Normalize path case for case-insensitive filesystems.

    On case-insensitive filesystems (Windows, macOS default), this ensures
    consistent casing. On case-sensitive systems (Linux), returns unchanged.

    Args:
        path: Path to normalize

    Returns:
        Path with normalized case
    """
    # Check if filesystem is case-sensitive
    # This is a heuristic - we check if we can create files differing only in case
    if path.exists():
        # Use actual case from filesystem
        try:
            # On Windows/macOS this will resolve to actual case
            return path.resolve()
        except Exception:
            pass

    return path


def resources(package: str, *path_parts: str) -> Path:
    """
    Access packaged resource files in a portable way.

    This uses importlib.resources (Python 3.9+) to access resources that
    are included in the installed package, whether from source or wheel.

    Args:
        package: Package name (e.g., "farfan_core.core")
        *path_parts: Path components within the package

    Returns:
        Path to the resource

    Raises:
        PathNotFoundError: If resource doesn't exist

    Example:
        >>> resources("farfan_core.core", "config", "default.yaml")
        Path('/path/to/farfan_core/core/config/default.yaml')
    """
    try:
        # Python 3.9+ way
        from importlib.resources import files
        pkg_path = files(package)
        for part in path_parts:
            pkg_path = pkg_path.joinpath(part)

        # Convert to Path - files() returns Traversable
        if hasattr(pkg_path, '__fspath__'):
            return Path(pkg_path)
        else:
            # Fallback for Traversable that doesn't support __fspath__
            # Read the resource and return a path to it
            raise PathNotFoundError(
                f"Resource '{'.'.join(path_parts)}' in package '{package}' "
                f"is not accessible as a filesystem path. "
                      f"Consider using importlib.resources.read_text() or read_binary()
instead."
            )
    except (ImportError, ModuleNotFoundError, FileNotFoundError, TypeError) as e:
        raise PathNotFoundError(
            f"Resource '{'/'.join(path_parts)}' not found in package '{package}'. "
            f"Ensure it's declared in pyproject.toml [tool.setuptools.package-data]. "
            f"Error: {e}"
        ) from e


def validate_read_path(path: Path) -> None:
    """
    Validate a path before reading from it.

    Args:
        path: Path to validate

    Raises:
        PathNotFoundError: If path doesn't exist
        PermissionError: If path is not readable
    """
    if not path.exists():
        raise PathNotFoundError(f"Path does not exist: '{path}'")

    if not os.access(path, os.R_OK):
        raise PermissionError(f"Path is not readable: '{path}'")


def validate_write_path(path: Path, allow_source_tree: bool = False) -> None:
    """
    Validate a path before writing to it.

    By default, prohibits writing to the source tree to prevent
    accidental modification of versioned code.

    Args:
        path: Path to validate
        allow_source_tree: If True, allow writing to source tree
                          (for special cases like code generation)
    Raises:
        PathOutsideWorkspaceError: If path is outside workspace
        PermissionError: If parent directory is not writable
        ValueError: If trying to write to source tree when not allowed
    """
    # Ensure it's within the workspace
    if not is_within(PROJECT_ROOT, path):
        raise PathOutsideWorkspaceError(
            f"Cannot write to '{path}' - outside workspace '{PROJECT_ROOT}'"
        )

    # Prohibit writing to source tree unless explicitly allowed
    if not allow_source_tree and is_within(SRC_DIR, path):
        raise ValueError(
            f"Cannot write to source tree: '{path}'. "
            f"Write to build/, cache/, or reports/ instead. "
            f"If you need to write to source (e.g., code generation), "
            f"set allow_source_tree=True."
        )

    # Ensure parent directory exists and is writable
    parent = path.parent
    if parent.exists() and not os.access(parent, os.W_OK):
        raise PermissionError(f"Parent directory is not writable: '{parent}'")


# Environment variable accessors (typed and safe)

def get_env_path(key: str, default: Path | None = None) -> Path | None:
    """
    Get a path from environment variable.

    Args:
        key: Environment variable name
        default: Default value if not set

    Returns:
        Path from environment or default
    """
    value = os.getenv(key)
    if value is None:
        return default
    return Path(value).resolve()


def get_workdir() -> Path:
    """
    Get the working directory from FLUX_WORKDIR env var or default to project root.
    """
    return get_env_path("FLUX_WORKDIR", PROJECT_ROOT) or PROJECT_ROOT


def get_tmpdir() -> Path:
```

```python
    """
    Get the temporary directory from FLUX_TMPDIR env var or default to project tmp.
    """
    result = get_env_path("FLUX_TMPDIR", tmp_dir()) or tmp_dir()
    result.mkdir(parents=True, exist_ok=True)
    return result


def get_reports_dir() -> Path:
    """
    Get the reports directory from FLUX_REPORTS env var or default to build/reports.
    """
    result = get_env_path("FLUX_REPORTS", reports_dir()) or reports_dir()
    result.mkdir(parents=True, exist_ok=True)
    return result


__all__ = [
    # Exceptions
    "PathError",
    "PathTraversalError",
    "PathNotFoundError",
    "PathOutsideWorkspaceError",
    "UnnormalizedPathError",
    # Constants
    "PROJECT_ROOT",
    "SRC_DIR",
    "DATA_DIR",
    "TESTS_DIR",
    "CONFIG_DIR",
    "QUESTIONNAIRE_FILE",
    # Directory accessors
    "proj_root",
    "src_dir",
    "data_dir",
    "tmp_dir",
    "build_dir",
    "cache_dir",
    "reports_dir",
    # Path operations
    "is_within",
    "safe_join",
    "normalize_unicode",
    "normalize_case",
    "resources",
    # Validation
    "validate_read_path",
    "validate_write_path",
    # Environment
    "get_env_path",
    "get_workdir",
    "get_tmpdir",
    "get_reports_dir",
]
```

src/farfan_pipeline/phases/Phase_zero/runtime_config.py

```
"""
Global runtime configuration system for F.A.R.F.A.N.

This module provides runtime mode enforcement (PROD/DEV/EXPLORATORY) with strict
fallback policies, configuration validation, and environment variable parsing.

FALLBACK CATEGORIZATION AND ASSESSMENT:

CATEGORY A (CRITICAL - System Integrity):
    Variables: ALLOW_CONTRADICTION_FALLBACK, ALLOW_VALIDATOR_DISABLE,
ALLOW_EXECUTION_ESTIMATES
    Assessment: These indicate missing CRITICAL components. In PROD, the system MUST
fail fast
    to prevent incorrect analysis results. No fallback is acceptable.

CATEGORY B (QUALITY - Quality Degradation):
    Variables: ALLOW_NETWORKX_FALLBACK, ALLOW_SPACY_FALLBACK
    Assessment: These degrade output quality but don't invalidate core analysis. Allowed
in
    PROD with explicit flag and warnings logged. Results remain scientifically valid but
less rich.

CATEGORY C (DEVELOPMENT - Development Convenience):
    Variables: ALLOW_DEV_INGESTION_FALLBACKS, ALLOW_AGGREGATION_DEFAULTS,
ALLOW_MISSING_BASE_WEIGHTS
    Assessment: STRICTLY FORBIDDEN in PROD. These exist only for development/testing to
avoid
    infrastructure dependencies. Using these in PROD invalidates results.

CATEGORY D (OPERATIONAL - Operational Flexibility):
    Variables: ALLOW_HASH_FALLBACK, ALLOW_PDFPLUMBER_FALLBACK
    Assessment: Safe fallbacks maintaining correctness with different implementation
strategies.
    Generally allowed as they don't affect scientific validity.

Environment Variables:
    SAAAAAA_RUNTIME_MODE: Runtime mode (prod/dev/exploratory), default: prod

    # Category A - Critical System Integrity
    ALLOW_CONTRADICTION_FALLBACK: Allow contradiction detection fallback, default: false
    ALLOW_VALIDATOR_DISABLE: Allow wiring validator disabling, default: false
    ALLOW_EXECUTION_ESTIMATES: Allow execution metric estimation, default: false

    # Category B - Quality Degradation
    ALLOW_NETWORKX_FALLBACK: Allow NetworkX unavailability, default: false
    ALLOW_SPACY_FALLBACK: Allow spaCy model fallback, default: false

    # Category C - Development Convenience (FORBIDDEN in PROD)
    ALLOW_DEV_INGESTION_FALLBACKS: Allow dev ingestion fallbacks, default: false
    ALLOW_AGGREGATION_DEFAULTS: Allow aggregation defaults, default: false

    # Category D - Operational Flexibility
    ALLOW_HASH_FALLBACK: Allow hash algorithm fallback, default: true
    ALLOW_PDFPLUMBER_FALLBACK: Allow pdfplumber unavailability, default: false

    # Model and Processing Configuration
    PREFERRED_SPACY_MODEL: Preferred spaCy model, default: es_core_news_lg
    PREFERRED_EMBEDDING_MODEL: Preferred embedding model, default:
sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2

    # Path Configuration
    SAAAAAA_PROJECT_ROOT: Project root override
    SAAAAAA_DATA_DIR: Data directory override
    SAAAAAA_OUTPUT_DIR: Output directory override
    SAAAAAA_CACHE_DIR: Cache directory override
    SAAAAAA_LOGS_DIR: Logs directory override

    # External Dependencies
    HF_ONLINE: Allow HuggingFace online access (0 or 1), default: 0

    # Processing Limits
    EXPECTED_QUESTION_COUNT: Expected question count, default: 305
    EXPECTED_METHOD_COUNT: Expected method count, default: 416
    PHASE_TIMEOUT_SECONDS: Phase timeout in seconds, default: 300
    MAX_WORKERS: Maximum worker threads, default: 4
    BATCH_SIZE: Batch size for processing, default: 100

Example:
    >>> config = RuntimeConfig.from_env()
    >>> if config.mode == RuntimeMode.PROD:
    ...     assert not config.allow_dev_ingestion_fallbacks
"""

import os
from dataclasses import dataclass
from enum import Enum
from typing import ClassVar, Optional


class RuntimeMode(Enum):
    """Runtime execution mode with different strictness levels."""

    PROD = "prod"
    """Production mode: strict enforcement, no fallbacks unless explicitly allowed."""

    DEV = "dev"
    """Development mode: permissive with flags, allows controlled degradation."""

    EXPLORATORY = "exploratory"
    """Exploratory mode: maximum flexibility for research and experimentation."""


class FallbackCategory(Enum):
    """Categorization of fallback types by impact."""

    CRITICAL = "critical"
    """Category A: System integrity - failures indicate missing critical dependencies."""

    QUALITY = "quality"
    """Category B: Quality degradation - system continues with reduced quality."""

    DEVELOPMENT = "development"
    """Category C: Development convenience - only allowed in DEV/EXPLORATORY."""

    OPERATIONAL = "operational"
    """Category D: Operational flexibility - safe fallbacks for operational concerns."""


class ConfigurationError(Exception):
    """Raised when runtime configuration is invalid or contains illegal combinations."""

    def __init__(self, message: str, illegal_combo: str | None = None) -> None:
        self.illegal_combo = illegal_combo
        super().__init__(message)


@dataclass(frozen=True)
class RuntimeConfig:
    """
    Immutable runtime configuration parsed from environment variables.

    This configuration controls system behavior across all components, enforcing
    strict policies in PROD mode and allowing controlled degradation in DEV/EXPLORATORY.

    Attributes:
        mode: Runtime execution mode

        # Category A - Critical System Integrity
        allow_contradiction_fallback: Allow fallback when contradiction module unavailable
        allow_validator_disable: Allow disabling wiring validator
        allow_execution_estimates: Allow execution metric estimation

        # Category B - Quality Degradation
        allow_networkx_fallback: Allow NetworkX unavailability
        allow_spacy_fallback: Allow spaCy model fallback

        # Category C - Development Convenience
        allow_dev_ingestion_fallbacks: Allow development ingestion fallbacks
        allow_aggregation_defaults: Allow aggregation default values
        allow_missing_base_weights: Allow missing base weights (legacy calibration flag)

        # Category D - Operational Flexibility
        allow_hash_fallback: Allow hash algorithm fallback
        allow_pdfplumber_fallback: Allow pdfplumber unavailability

        # Model Configuration
        preferred_spacy_model: Preferred spaCy model name
        preferred_embedding_model: Preferred embedding model name
        # Path Configuration
        project_root_override: Project root path override
        data_dir_override: Data directory override
        output_dir_override: Output directory override
        cache_dir_override: Cache directory override
        logs_dir_override: Logs directory override

        # External Dependencies
        hf_online: Allow HuggingFace online access

        # Processing Configuration
        expected_question_count: Expected question count for validation
        expected_method_count: Expected method count for validation
        phase_timeout_seconds: Phase timeout in seconds
        max_workers: Maximum worker threads
        batch_size: Batch size for processing
    """

    mode: RuntimeMode

    # Category A - Critical
    allow_contradiction_fallback: bool
    allow_validator_disable: bool
    allow_execution_estimates: bool

    # Category B - Quality
    allow_networkx_fallback: bool
    allow_spacy_fallback: bool

    # Category C - Development
    allow_dev_ingestion_fallbacks: bool
    allow_aggregation_defaults: bool
    allow_missing_base_weights: bool

    # Category D - Operational
    allow_hash_fallback: bool
    allow_pdfplumber_fallback: bool

    # Model Configuration
    preferred_spacy_model: str
    preferred_embedding_model: str

    # Path Configuration
    project_root_override: Optional[str]
    data_dir_override: Optional[str]
    output_dir_override: Optional[str]
    cache_dir_override: Optional[str]
    logs_dir_override: Optional[str]

    # External Dependencies
    hf_online: bool

    # Processing Configuration
```

```
    expected_question_count: int
    expected_method_count: int
    phase_timeout_seconds: int
    max_workers: int
    batch_size: int

    # Illegal combinations in PROD mode
    _PROD_ILLEGAL_COMBOS: ClassVar[dict[str, tuple[str, FallbackCategory]]] = {
        "ALLOW_DEV_INGESTION_FALLBACKS": (
            "Development ingestion fallbacks not allowed in PROD - they bypass quality
gates",
            FallbackCategory.DEVELOPMENT
        ),
        "ALLOW_EXECUTION_ESTIMATES": (
            "Execution metric estimation not allowed in PROD - actual measurements
required",
            FallbackCategory.CRITICAL
        ),
        "ALLOW_AGGREGATION_DEFAULTS": (
            "Aggregation defaults not allowed in PROD - explicit calibration required",
            FallbackCategory.DEVELOPMENT
        ),
        "ALLOW_MISSING_BASE_WEIGHTS": (
            "Missing base weights not allowed in PROD - complete calibration required",
            FallbackCategory.DEVELOPMENT,
        ),
    }

    @classmethod
    def from_dict(cls, data: dict) -> "RuntimeConfig":
        """
        Create RuntimeConfig from dictionary (for testing).

        Args:
            data: Dictionary with configuration values

        Returns:
            RuntimeConfig: Configuration instance
        """
        mode_val = data.get("mode", "prod")
        if isinstance(mode_val, RuntimeMode):
            mode = mode_val
        else:
            try:
                mode = RuntimeMode(str(mode_val).lower())
            except ValueError:
                mode = RuntimeMode.PROD

        return cls(
            mode=mode,
            allow_contradiction_fallback=data.get("allow_contradiction_fallback",
False),
            allow_validator_disable=data.get("allow_validator_disable", False),
            allow_execution_estimates=data.get("allow_execution_estimates", False),
            allow_networkx_fallback=data.get("allow_networkx_fallback", False),
            allow_spacy_fallback=data.get("allow_spacy_fallback", False),
            allow_dev_ingestion_fallbacks=data.get("allow_dev_ingestion_fallbacks",
False),
            allow_aggregation_defaults=data.get("allow_aggregation_defaults", False),
            allow_missing_base_weights=data.get("allow_missing_base_weights", False),
            allow_hash_fallback=data.get("allow_hash_fallback", True),
            allow_pdfplumber_fallback=data.get("allow_pdfplumber_fallback", False),
            preferred_spacy_model=data.get("preferred_spacy_model", "es_core_news_lg"),
            preferred_embedding_model=data.get("preferred_embedding_model",
"sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"),
            project_root_override=data.get("project_root_override"),
            data_dir_override=data.get("data_dir_override"),
            output_dir_override=data.get("output_dir_override"),
            cache_dir_override=data.get("cache_dir_override"),
            logs_dir_override=data.get("logs_dir_override"),
            hf_online=data.get("hf_online", False),
            expected_question_count=data.get("expected_question_count", 305),
            expected_method_count=data.get("expected_method_count", 416),
            phase_timeout_seconds=data.get("phase_timeout_seconds", 300),
            max_workers=data.get("max_workers", 4),
            batch_size=data.get("batch_size", 100),
        )

    @classmethod
    def from_env(cls) -> "RuntimeConfig":
        """
        Parse runtime configuration from environment variables.

        Returns:
            RuntimeConfig: Validated configuration instance

        Raises:
            ConfigurationError: If configuration is invalid or contains illegal
combinations

        Example:
            >>> os.environ['SAAAAAA_RUNTIME_MODE'] = 'prod'
            >>> config = RuntimeConfig.from_env()
            >>> assert config.mode == RuntimeMode.PROD
        """
        # Parse runtime mode
        mode_str = os.getenv("SAAAAAA_RUNTIME_MODE", "prod").lower()
        try:
            mode = RuntimeMode(mode_str)
        except ValueError as e:
            raise ConfigurationError(
                f"Invalid SAAAAAA_RUNTIME_MODE: {mode_str}. "
                f"Must be one of: {', '.join(m.value for m in RuntimeMode)}"
            ) from e

        # Parse Category A - Critical Fallbacks
        allow_contradiction_fallback = _parse_bool_env("ALLOW_CONTRADICTION_FALLBACK",
False)
```

```
        allow_validator_disable = _parse_bool_env("ALLOW_VALIDATOR_DISABLE", False)
        allow_execution_estimates = _parse_bool_env("ALLOW_EXECUTION_ESTIMATES", False)

        # Parse Category B - Quality Fallbacks
        allow_networkx_fallback = _parse_bool_env("ALLOW_NETWORKX_FALLBACK", False)
        allow_spacy_fallback = _parse_bool_env("ALLOW_SPACY_FALLBACK", False)

        # Parse Category C - Development Fallbacks
         allow_dev_ingestion_fallbacks = _parse_bool_env("ALLOW_DEV_INGESTION_FALLBACKS",
False)
             allow_aggregation_defaults  =  _parse_bool_env("ALLOW_AGGREGATION_DEFAULTS",
False)
             allow_missing_base_weights  =  _parse_bool_env("ALLOW_MISSING_BASE_WEIGHTS",
False)

        # Parse Category D - Operational Fallbacks
        allow_hash_fallback = _parse_bool_env("ALLOW_HASH_FALLBACK", True)
        allow_pdfplumber_fallback = _parse_bool_env("ALLOW_PDFPLUMBER_FALLBACK", False)

        # Parse model configuration
        preferred_spacy_model = os.getenv("PREFERRED_SPACY_MODEL", "es_core_news_lg")
        preferred_embedding_model = os.getenv(
            "PREFERRED_EMBEDDING_MODEL",
            "sentence-transformers/paraphrase-multilingual-MiniLM-L12-v2"
        )

        # Parse path configuration
        project_root_override = os.getenv("SAAAAAA_PROJECT_ROOT")
        data_dir_override = os.getenv("SAAAAAA_DATA_DIR")
        output_dir_override = os.getenv("SAAAAAA_OUTPUT_DIR")
        cache_dir_override = os.getenv("SAAAAAA_CACHE_DIR")
        logs_dir_override = os.getenv("SAAAAAA_LOGS_DIR")

        # Parse external dependencies
        hf_online = os.getenv("HF_ONLINE", "0") == "1"

        # Parse processing configuration
        expected_question_count = _parse_int_env("EXPECTED_QUESTION_COUNT", 305)
        expected_method_count = _parse_int_env("EXPECTED_METHOD_COUNT", 416)
        phase_timeout_seconds = _parse_int_env("PHASE_TIMEOUT_SECONDS", 300)
        max_workers = _parse_int_env("MAX_WORKERS", 4)
        batch_size = _parse_int_env("BATCH_SIZE", 100)

        # Create config instance
        config = cls(
            mode=mode,
            allow_contradiction_fallback=allow_contradiction_fallback,
            allow_validator_disable=allow_validator_disable,
            allow_execution_estimates=allow_execution_estimates,
            allow_networkx_fallback=allow_networkx_fallback,
            allow_spacy_fallback=allow_spacy_fallback,
            allow_dev_ingestion_fallbacks=allow_dev_ingestion_fallbacks,
            allow_aggregation_defaults=allow_aggregation_defaults,
            allow_missing_base_weights=allow_missing_base_weights,
            allow_hash_fallback=allow_hash_fallback,
            allow_pdfplumber_fallback=allow_pdfplumber_fallback,
            preferred_spacy_model=preferred_spacy_model,
            preferred_embedding_model=preferred_embedding_model,
            project_root_override=project_root_override,
            data_dir_override=data_dir_override,
            output_dir_override=output_dir_override,
            cache_dir_override=cache_dir_override,
            logs_dir_override=logs_dir_override,
            hf_online=hf_online,
            expected_question_count=expected_question_count,
            expected_method_count=expected_method_count,
            phase_timeout_seconds=phase_timeout_seconds,
            max_workers=max_workers,
            batch_size=batch_size,
        )

        # Validate configuration
        config._validate()

        return config

    def _validate(self) -> None:
        """
        Validate configuration for illegal combinations.

        In PROD mode, certain ALLOW_* flags are prohibited to ensure strict behavior.

        Raises:
            ConfigurationError: If illegal combination detected
        """
        if self.mode != RuntimeMode.PROD:
            return  # DEV/EXPLORATORY modes allow all combinations

        # Check for illegal PROD combinations
        violations = []

        if self.allow_dev_ingestion_fallbacks:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_DEV_INGESTION_FALLBACKS"]
            violations.append(
                f"PROD + ALLOW_DEV_INGESTION_FALLBACKS=true: {msg} [Category:
{cat.value}]"
            )

        if self.allow_execution_estimates:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_EXECUTION_ESTIMATES"]
            violations.append(
                f"PROD + ALLOW_EXECUTION_ESTIMATES=true: {msg} [Category: {cat.value}]"
            )

        if self.allow_aggregation_defaults:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_AGGREGATION_DEFAULTS"]
            violations.append(
                f"PROD + ALLOW_AGGREGATION_DEFAULTS=true: {msg} [Category: {cat.value}]"
            )

        if self.allow_missing_base_weights:
            msg, cat = self._PROD_ILLEGAL_COMBOS["ALLOW_MISSING_BASE_WEIGHTS"]
            violations.append(
                f"PROD + ALLOW_MISSING_BASE_WEIGHTS=true: {msg} [Category: {cat.value}]"
            )

        if violations:
            raise ConfigurationError(
                 "Illegal configuration combinations detected:\n" + "\n".join(f"  - {v}"
for v in violations),
                illegal_combo="; ".join(violations)
            )

    def is_strict_mode(self) -> bool:
        """Check if running in strict mode (PROD with no fallbacks allowed)."""
        return (
            self.mode == RuntimeMode.PROD
            and not self.allow_contradiction_fallback
            and not self.allow_validator_disable
        )

    @property
    def strict_calibration(self) -> bool:
        """
        Check if strict calibration is required.

        In PROD mode, strict calibration is enforced unless explicitly relaxed.
        This means no missing base weights are allowed.

        Returns:
                            True if strict calibration is required (PROD without
allow_missing_base_weights)
        """
        return self.mode == RuntimeMode.PROD and not self.allow_missing_base_weights

    def get_fallback_summary(self) -> dict[str, dict[str, bool]]:
        """
        Get summary of all fallback configurations grouped by category.

        Returns:
            Dictionary mapping category names to flag dictionaries
        """
        return {
            "critical": {
                "contradiction_fallback": self.allow_contradiction_fallback,
                "validator_disable": self.allow_validator_disable,
                "execution_estimates": self.allow_execution_estimates,
            },
            "quality": {
                "networkx_fallback": self.allow_networkx_fallback,
                "spacy_fallback": self.allow_spacy_fallback,
            },
            "development": {
                "dev_ingestion_fallbacks": self.allow_dev_ingestion_fallbacks,
                "aggregation_defaults": self.allow_aggregation_defaults,
                "missing_base_weights": self.allow_missing_base_weights,
            },
            "operational": {
                "hash_fallback": self.allow_hash_fallback,
                "pdfplumber_fallback": self.allow_pdfplumber_fallback,
            },
        }

    def __repr__(self) -> str:
        """String representation showing mode and key flags."""
        flags = []
        if self.allow_contradiction_fallback:
            flags.append("contradiction_fallback")
        if self.allow_validator_disable:
            flags.append("validator_disable")
        if self.allow_execution_estimates:
            flags.append("execution_estimates")
        if self.allow_networkx_fallback:
            flags.append("networkx_fallback")
        if self.allow_spacy_fallback:
            flags.append("spacy_fallback")
        if self.allow_dev_ingestion_fallbacks:
            flags.append("dev_ingestion_fallbacks")
        if self.allow_aggregation_defaults:
            flags.append("aggregation_defaults")
        if self.allow_missing_base_weights:
            flags.append("missing_base_weights")
        if not self.strict_calibration:
            flags.append("relaxed_calibration")

        flags_str = f", flags={flags}" if flags else ""
        return f"RuntimeConfig(mode={self.mode.value}{flags_str})"


def _parse_bool_env(var_name: str, default: bool) -> bool:
    """
    Parse boolean environment variable with case-insensitive handling.

    Args:
        var_name: Environment variable name
        default: Default value if not set

    Returns:
        Parsed boolean value

    Raises:
        ConfigurationError: If value is not a valid boolean
    """
    value = os.getenv(var_name)
    if value is None:
        return default
    value_lower = value.lower()
    if value_lower in ("true", "1", "yes", "on"):
        return True
    elif value_lower in ("false", "0", "no", "off"):
        return False
    else:
        raise ConfigurationError(
            f"Invalid boolean value for {var_name}: {value}. "
            f"Must be one of: true/false, 1/0, yes/no, on/off"
        )


def _parse_int_env(var_name: str, default: int) -> int:
    """
    Parse integer environment variable with validation.

    Args:
        var_name: Environment variable name
        default: Default value if not set

    Returns:
        Parsed integer value

    Raises:
        ConfigurationError: If value is not a valid integer
    """
    value = os.getenv(var_name)
    if value is None:
        return default

    try:
        return int(value)
    except ValueError:
        raise ConfigurationError(
            f"Invalid integer value for {var_name}: {value}. "
            f"Must be a valid integer."
        )


# Global singleton instance (lazy-initialized)
_global_config: RuntimeConfig | None = None


def get_runtime_config() -> RuntimeConfig:
    """
    Get global runtime configuration instance (lazy-initialized).

    Returns:
        RuntimeConfig: Global configuration instance

    Note:
        This is initialized once on first call. For testing, use from_env() directly.
    """
    global _global_config
    if _global_config is None:
        _global_config = RuntimeConfig.from_env()
    return _global_config


def reset_runtime_config() -> None:
    """
    Reset global runtime configuration (for testing only).

    Warning:
        This should only be used in tests. Production code should never reset config.
    """
    global _global_config
    _global_config = None
```

src/farfan_pipeline/phases/Phase_zero/runtime_error_fixes.py

```python
"""
Runtime Error Fixes for Policy Analysis

This module contains fixes for three critical runtime errors:
1. 'bool' object is not iterable - Functions returning bool instead of list
2. 'str' object has no attribute 'text' - String passed where spacy object expected
3. can't multiply sequence by non-int of type 'float' - List multiplication by float

These fixes are applied defensively to prevent crashes in production.
"""

from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    import numpy as np
    NumpyArray = np.ndarray
else:
    NumpyArray = Any  # type: ignore[misc]

try:
    HAS_NUMPY = True
except ImportError:
    HAS_NUMPY = False

def ensure_list_return(value: Any) -> list[Any]:
    """
    Ensure a value is a list, converting bool/None to empty list.

    Fixes: 'bool' object is not iterable

    Args:
        value: Value that should be a list

    Returns:
        Empty list if value is False/None/bool, otherwise the value as-is
    """
    if isinstance(value, bool) or value is None:
        return []
    if isinstance(value, list):
        return value
    # If it's iterable but not a list, convert it
    try:
        return list(value)
    except (TypeError, ValueError):
        return []

def safe_text_extract(obj: Any) -> str:
    """
    Safely extract text from object that might be str or have .text attribute.

    Fixes: 'str' object has no attribute 'text'
    Args:
        obj: Object that is either str or has .text attribute (e.g., spacy Doc/Span)

    Returns:
        Extracted text string
    """
    # If it's already a string, return it
    if isinstance(obj, str):
        return obj

    # If it has a .text attribute, extract it
    if hasattr(obj, 'text'):
        text_value = obj.text
        if isinstance(text_value, str):
            return text_value

    # Fallback: convert to string
    return str(obj)

def safe_weighted_multiply(items: list[float] | NumpyArray, weight: float) ->
list[float] | NumpyArray:
    """
    Safely multiply a list or array by a weight.

    Fixes: can't multiply sequence by non-int of type 'float'

    Args:
        items: List or array of numbers
        weight: Weight to multiply by

    Returns:
        New list/array with each element multiplied by weight
    """
    # If it's a numpy array, use numpy multiplication
    if HAS_NUMPY and hasattr(items, '__array_interface__'):
        import numpy as np  # Import here for runtime use
        if isinstance(items, np.ndarray):
            return items * weight

    # If it's a list, use list comprehension
    if isinstance(items, list):
        return [item * weight for item in items]

    # If it's something else iterable, convert and multiply
    try:
        return [item * weight for item in items]
    except (TypeError, ValueError):
        # If multiplication fails, return empty list
        return []

def safe_list_iteration(value: Any) -> list[Any]:
    """
    Ensure a value can be safely iterated over.
```

```
Converts bool, None, or non-iterables to empty list.
Handles the common error of trying to iterate over bool.

Args:
    value: Value to iterate over

Returns:
    Iterable list
"""
# Reject booleans explicitly
if isinstance(value, bool):
    return []

# Handle None
if value is None:
    return []

# If it's already a list, return it
if isinstance(value, list):
    return value

# If it's a string, don't iterate over characters - return as single item
if isinstance(value, str):
    return [value]

# Try to convert to list
try:
    return list(value)
except (TypeError, ValueError):
    return []
```

src/farfan_pipeline/phases/Phase_zero/schema_monitor.py

```
"""
SCHEMA DRIFT MONITORING - Watch Production Payloads
===================================================

Sample payloads in staging/prod and validate shapes.
Emit metrics on key presence/type.
Page when new keys appear or required keys vanish.

Catches upstream changes (or LLM output drift) instantly.
"""

from __future__ import annotations

import json
import logging
import random
from collections import Counter, defaultdict
from dataclasses import dataclass, field
from datetime import datetime
from typing import TYPE_CHECKING, Any, TypedDict
from farfan_pipeline.core.parameters import ParameterLoaderV2

if TYPE_CHECKING:
    from collections.abc import Mapping
    from pathlib import Path

logger = logging.getLogger(__name__)

# ============================================================================
# SCHEMA SHAPE TRACKING
# ============================================================================

class SchemaShape(TypedDict):
    """Shape of a data payload."""

    keys: set[str]
    types: dict[str, str]
    sample_values: dict[str, Any]
    timestamp: str

@dataclass
class SchemaStats:
    """Statistics about schema shape over time."""

    key_frequency: Counter[str] = field(default_factory=Counter)
            type_by_key:   dict[str,   Counter[str]]   =   field(default_factory=lambda:
defaultdict(Counter))
    new_keys: set[str] = field(default_factory=set)
    missing_keys: set[str] = field(default_factory=set)
    total_samples: int = 0
    last_updated: datetime | None = None
```

```python
class SchemaDriftDetector:
    """
    Detects schema drift by sampling payloads and tracking shape changes.

    Usage:
        detector = SchemaDriftDetector(sample_rate=0.05)

        # In your API/pipeline
        if detector.should_sample():
            detector.record_payload(data, source="api_input")

        # Check for drift
        alerts = detector.get_alerts()
    """

    def __init__(
        self,
        *,
        sample_rate: float = 0.05,
        baseline_path: Path | None = None,
        alert_threshold: float = 0.1,
    ) -> None:
        """
        Initialize schema drift detector.

        Args:
            sample_rate: Percentage of payloads to sample (0.01 = 1%, 0.05 = 5%)
            baseline_path: Path to baseline schema file
            alert_threshold: Threshold for drift alert (% of samples with drift)
        """
        self.sample_rate = sample_rate
        self.baseline_path = baseline_path
        self.alert_threshold = alert_threshold

        # Tracking state
        self.stats_by_source: dict[str, SchemaStats] = defaultdict(SchemaStats)
        self.baseline_schema: dict[str, SchemaShape] = {}

        # Load baseline if provided
        if baseline_path and baseline_path.exists():
            self._load_baseline()

    def should_sample(self) -> bool:
        """Decide whether to sample this payload (probabilistic)."""
        return random.random() < self.sample_rate

    def record_payload(
        self,
        payload: Mapping[str, Any],
        *,
        source: str,
        timestamp: datetime | None = None,
    ) -> None:
        """
        Record a payload for schema tracking.

        Args:
            payload: Data payload to analyze
            source: Source identifier (e.g., "api_input", "document_loader")
            timestamp: Optional timestamp, defaults to now
        """
        if timestamp is None:
            timestamp = datetime.utcnow()

        stats = self.stats_by_source[source]

        # Extract shape
        keys = set(payload.keys())
        types = {k: type(v).__name__ for k, v in payload.items()}

        # Update statistics
        stats.total_samples += 1
        stats.last_updated = timestamp

        for key in keys:
            stats.key_frequency[key] += 1
            stats.type_by_key[key][types[key]] += 1

        # Detect new keys (compared to baseline)
        if source in self.baseline_schema:
            baseline_keys = self.baseline_schema[source]["keys"]
            new_keys = keys - baseline_keys
            if new_keys:
                stats.new_keys.update(new_keys)
                logger.warning(
                    f"SCHEMA_DRIFT[source={source}]: New keys detected: {new_keys}"
                )

            missing_keys = baseline_keys - keys
            if missing_keys:
                stats.missing_keys.update(missing_keys)
                logger.warning(
                    f"SCHEMA_DRIFT[source={source}]: Missing keys detected:
{missing_keys}"
                )

    def get_alerts(self, *, source: str | None = None) -> list[dict[str, Any]]:
        """
        Get schema drift alerts.

        Args:
            source: Optional source filter

        Returns:
            List of alert dicts
        """
        alerts: list[dict[str, Any]] = []
        sources = [source] if source else list(self.stats_by_source.keys())

        for src in sources:
            stats = self.stats_by_source[src]

            if stats.new_keys:
                alerts.append({
                    "level": "WARNING",
                    "source": src,
                    "type": "NEW_KEYS",
                    "keys": list(stats.new_keys),
                       "timestamp": stats.last_updated.isoformat() if stats.last_updated
else None,
                })

            if stats.missing_keys:
                alerts.append({
                    "level": "CRITICAL",
                    "source": src,
                    "type": "MISSING_KEYS",
                    "keys": list(stats.missing_keys),
                       "timestamp": stats.last_updated.isoformat() if stats.last_updated
else None,
                })

            # Check for type inconsistencies
            for key, type_counts in stats.type_by_key.items():
                if len(type_counts) > 1:
                    # Multiple types seen for same key
                    dominant_type = type_counts.most_common(1)[0][0]
                    other_types = [t for t in type_counts if t != dominant_type]

                    alerts.append({
                        "level": "WARNING",
                        "source": src,
                        "type": "TYPE_INCONSISTENCY",
                        "key": key,
                        "expected_type": dominant_type,
                        "observed_types": other_types,
                                         "timestamp": stats.last_updated.isoformat() if
stats.last_updated else None,
                    })

        return alerts

    def save_baseline(self, output_path: Path) -> None:
        """
        Save current schema shapes as baseline.

        Args:
            output_path: Path to save baseline JSON
        """
        baseline: dict[str, dict[str, Any]] = {}
        for source, stats in self.stats_by_source.items():
            # Get most common keys (present in >50% of samples)
                                               threshold    =    stats.total_samples    *
ParameterLoaderV2.get("farfan_core.utils.schema_monitor.SchemaDriftDetector.save_baselin
e", "auto_param_L215_46", 0.5)
            common_keys = {
                key for key, count in stats.key_frequency.items()
                if count >= threshold
            }

            # Get dominant type for each key
            types = {
                key: type_counts.most_common(1)[0][0]
                for key, type_counts in stats.type_by_key.items()
            }

            baseline[source] = {
                "keys": list(common_keys),
                "types": types,
                "timestamp": datetime.utcnow().isoformat(),
            }

        output_path.write_text(json.dumps(baseline, indent=2))
        logger.info(f"Saved schema baseline to {output_path}")

    def _load_baseline(self) -> None:
        """Load baseline schema from file."""
        if not self.baseline_path:
            return

        try:
            data = json.loads(self.baseline_path.read_text())

            for source, shape_data in data.items():
                self.baseline_schema[source] = {
                    "keys": set(shape_data["keys"]),
                    "types": shape_data["types"],
                    "sample_values": {},
                    "timestamp": shape_data["timestamp"],
                }

            logger.info(f"Loaded schema baseline from {self.baseline_path}")
        except Exception as e:
            logger.error(f"Failed to load baseline: {e}")

    def get_metrics(self, *, source: str | None = None) -> dict[str, Any]:
        """
        Get monitoring metrics.

        Args:
            source: Optional source filter

        Returns:
            Dict of metrics
        """
        if source:
            stats = self.stats_by_source.get(source)
            if not stats:
                return {}

            return {
                "source": source,
                "total_samples": stats.total_samples,
                "unique_keys": len(stats.key_frequency),
                "new_keys_count": len(stats.new_keys),
                "missing_keys_count": len(stats.missing_keys),
                "type_inconsistencies": sum(
                    1 for counts in stats.type_by_key.values()
                    if len(counts) > 1
                ),
            }

        # Aggregate across all sources
        return {
            "sources": list(self.stats_by_source.keys()),
            "total_samples": sum(s.total_samples for s in
self.stats_by_source.values()),
            "sources_with_drift": len([
                s for s in self.stats_by_source.values()
                if s.new_keys or s.missing_keys
            ]),
        }

# ============================================================================
# PAYLOAD VALIDATOR
# ============================================================================

class PayloadValidator:
    """
    Validate payloads against expected schema.

    Usage:
        validator = PayloadValidator(schema_path=Path("schemas/api_input.json"))

        try:
            validator.validate(data, source="api_endpoint")
        except ValueError as e:
            logger.error(f"Validation failed: {e}")
    """

    def __init__(self, *, schema_path: Path | None = None) -> None:
        """
        Initialize payload validator.

        Args:
            schema_path: Path to schema definition JSON
        """
        self.schema_path = schema_path
```

```python
        self.schemas: dict[str, dict[str, Any]] = {}

        if schema_path and schema_path.exists():
            self._load_schemas()

    def validate(
        self,
        payload: Mapping[str, Any],
        *,
        source: str,
        strict: bool = True,
    ) -> None:
        """
        Validate payload against schema.

        Args:
            payload: Data payload to validate
            source: Source identifier
            strict: If True, raise on missing keys; if False, only warn

        Raises:
            ValueError: If validation fails in strict mode
            TypeError: If value types don't match schema
        """
        if source not in self.schemas:
            logger.warning(f"No schema defined for source '{source}'")
            return

        schema = self.schemas[source]
        required_keys = set(schema.get("required_keys", []))
        expected_types = schema.get("types", {})

        # Check required keys
        payload_keys = set(payload.keys())
        missing = required_keys - payload_keys

        if missing:
            msg = f"VALIDATION_ERROR[source={source}]: Missing required keys: {missing}"
            if strict:
                raise ValueError(msg)
            else:
                logger.warning(msg)

        # Check types
        for key, expected_type in expected_types.items():
            if key in payload:
                actual_type = type(payload[key]).__name__
                if actual_type != expected_type:
                    msg = (
                        f"VALIDATION_ERROR[source={source}, key={key}]: "
                        f"Expected type {expected_type}, got {actual_type}"
                    )
                    if strict:
                        raise TypeError(msg)
                    else:
                        logger.warning(msg)

    def _load_schemas(self) -> None:
        """Load schema definitions from file."""
        if not self.schema_path:
            return

        try:
            self.schemas = json.loads(self.schema_path.read_text())
            logger.info(f"Loaded schemas from {self.schema_path}")
        except Exception as e:
            logger.error(f"Failed to load schemas: {e}")

# ===========================================================================
# GLOBAL INSTANCE (optional convenience)
# ===========================================================================

# Singleton detector for application-wide use
_global_detector: SchemaDriftDetector | None = None

def get_detector() -> SchemaDriftDetector:
    """Get or create global schema drift detector."""
    global _global_detector
    if _global_detector is None:
                                                                 _global_detector        =
SchemaDriftDetector(sample_rate=ParameterLoaderV2.get("farfan_core.utils.schema_monitor.
PayloadValidator._load_schemas", "auto_param_L400_59", 0.05))
    return _global_detector
```

src/farfan_pipeline/phases/Phase_zero/seed_factory.py

```
"""
Deterministic Seed Factory
Generates reproducible seeds for all stochastic operations
"""

import hashlib
import hmac
import random
from typing import Any

try:
    import numpy as np
    NUMPY_AVAILABLE = True
except ImportError:
    NUMPY_AVAILABLE = False
    np = None  # type: ignore

class SeedFactory:
    """
    Factory for generating deterministic seeds

    Ensures:
    - Reproducibility: Same inputs ? same seed
    - Uniqueness: Different contexts ? different seeds
    - Cryptographic quality: HMAC-SHA256 derivation
    """

    # Fixed salt for seed derivation (should be configured per deployment)
    DEFAULT_SALT = b"PDM_EVALUATOR_V2_DETERMINISTIC_SEED_2025"

    def __init__(self, fixed_salt: bytes | None = None) -> None:
        self.salt = fixed_salt or self.DEFAULT_SALT

    def create_deterministic_seed(
        self,
        correlation_id: str,
        file_checksums: dict[str, str] | None = None,
        context: dict[str, Any] | None = None
    ) -> int:
        """
        Generate deterministic seed from correlation ID and context

        Args:
            correlation_id: Unique workflow instance identifier
            file_checksums: Dict of {filename: sha256_checksum}
            context: Additional context (question_id, policy_area, etc.)

        Returns:
            32-bit integer seed (0 to 2^32-1)

        Example:
            >>> factory = SeedFactory()
            >>> seed1 = factory.create_deterministic_seed("run-001", {"data.json":
"abc123"})
            >>> seed2 = factory.create_deterministic_seed("run-001", {"data.json":
"abc123"})
            >>> assert seed1 == seed2  # Reproducible
        """

        # Build deterministic input string
        components = [correlation_id]

        # Add file checksums (sorted for determinism)
        if file_checksums:
            sorted_checksums = sorted(file_checksums.items())
            for filename, checksum in sorted_checksums:
                components.append(f"{filename}:{checksum}")

        # Add context (sorted for determinism)
        if context:
            sorted_context = sorted(context.items())
            for key, value in sorted_context:
                components.append(f"{key}={value}")

        # Combine with deterministic separator
        seed_input = "|".join(components).encode('utf-8')

        # HMAC-SHA256 for cryptographic quality
        seed_hmac = hmac.new(
            key=self.salt,
            msg=seed_input,
            digestmod=hashlib.sha256
        )

        # Convert to 32-bit integer
        seed_bytes = seed_hmac.digest()[:4]  # First 4 bytes
        seed_int = int.from_bytes(seed_bytes, byteorder='big')

        return seed_int

    def configure_global_random_state(self, seed: int) -> None:
        """
        Configure all random number generators with seed

        Sets:
        - Python random module
        - NumPy random state
        - (Add torch, tensorflow if needed)

        Args:
            seed: Deterministic seed
        """

        # Python random module
        random.seed(seed)
        # NumPy
        if NUMPY_AVAILABLE and np is not None:
            np.random.seed(seed)

        # TODO: Add torch.manual_seed(seed) if PyTorch is used
        # TODO: Add tf.random.set_seed(seed) if TensorFlow is used

class DeterministicContext:
    """
    Context manager for deterministic execution

    Usage:
        with DeterministicContext(correlation_id="run-001") as seed:
            # All random operations are deterministic
            result = some_stochastic_function()
    """

    def __init__(
        self,
        correlation_id: str,
        file_checksums: dict[str, str] | None = None,
        context: dict[str, Any] | None = None,
        fixed_salt: bytes | None = None
    ) -> None:
        self.correlation_id = correlation_id
        self.file_checksums = file_checksums
        self.context = context
        self.factory = SeedFactory(fixed_salt=fixed_salt)

        self.seed: int | None = None
        self.previous_random_state = None
        self.previous_numpy_state = None

    def __enter__(self) -> int:
        """Enter deterministic context"""

        # Generate deterministic seed
        self.seed = self.factory.create_deterministic_seed(
            correlation_id=self.correlation_id,
            file_checksums=self.file_checksums,
            context=self.context
        )

        # Save current random states
        self.previous_random_state = random.getstate()
        if NUMPY_AVAILABLE and np is not None:
            self.previous_numpy_state = np.random.get_state()

        # Configure with deterministic seed
        self.factory.configure_global_random_state(self.seed)

        return self.seed

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Exit deterministic context and restore previous state"""

        # Restore previous random states
        if self.previous_random_state:
            random.setstate(self.previous_random_state)

        if NUMPY_AVAILABLE and np is not None and self.previous_numpy_state:
            np.random.set_state(self.previous_numpy_state)

        return False

def create_deterministic_seed(
    correlation_id: str,
    file_checksums: dict[str, str] | None = None,
    **context_kwargs
) -> int:
    """
    Convenience function for creating deterministic seed

    Args:
        correlation_id: Unique workflow instance ID
        file_checksums: Dict of file checksums
        **context_kwargs: Additional context as keyword arguments

    Returns:
        Deterministic 32-bit integer seed

    Example:
        >>> seed = create_deterministic_seed(
        ...     "run-001",
        ...     question_id="Q001",
        ...     policy_area_id="PA01"
        ... )
    """
    factory = SeedFactory()
    return factory.create_deterministic_seed(
        correlation_id=correlation_id,
        file_checksums=file_checksums,
        context=context_kwargs if context_kwargs else None
    )
```

src/farfan_pipeline/phases/Phase_zero/signature_validator.py

```
"""
Signature Validation and Interface Governance System
====================================================

Implements automated signature consistency auditing, runtime validation,
and interface governance to prevent function signature mismatches.

Based on the Strategic Mitigation Plan for addressing interface inconsistencies.

Author: Signature Governance Team
Version: 1.0.0
"""

import ast
import functools
import hashlib
import inspect
import json
import logging
from collections.abc import Callable
from dataclasses import asdict, dataclass, field
from datetime import datetime
from pathlib import Path
from typing import Any, TypeVar, get_type_hints

logger = logging.getLogger(__name__)

# Type variable for decorated functions
F = TypeVar('F', bound=Callable[..., Any])

# ============================================================================
# SIGNATURE METADATA STORAGE
# ============================================================================

@dataclass
class FunctionSignature:
    """Stores metadata about a function's signature"""
    module: str
    class_name: str | None
    function_name: str
    parameters: list[str]
    parameter_types: dict[str, str]
    return_type: str
    signature_hash: str
    timestamp: str = field(default_factory=lambda: datetime.now().isoformat())

    def to_dict(self) -> dict[str, Any]:
        return asdict(self)

class SignatureRegistry:
    """
    Maintains a registry of function signatures with version tracking
    Implements signature snapshotting as described in the mitigation plan
    """

    def __init__(self, registry_path: Path = Path("data/signature_registry.json")) ->
None:
        self.registry_path = registry_path
        self.signatures: dict[str, FunctionSignature] = {}
        self.load()

    def compute_signature_hash(self, func: Callable) -> str:
        """Compute a hash of the function's signature"""
        sig = inspect.signature(func)
        sig_str = str(sig)
        return hashlib.sha256(sig_str.encode()).hexdigest()[:16]

    def register_function(self, func: Callable) -> FunctionSignature:
        """Register a function's signature"""
        sig = inspect.signature(func)

        # Extract parameter information
        parameters = list(sig.parameters.keys())

        # Get type hints if available
        try:
            type_hints = get_type_hints(func)
            parameter_types = {
                name: str(type_hints.get(name, 'Any'))
                for name in parameters
            }
            return_type = str(type_hints.get('return', 'Any'))
        except (TypeError, AttributeError, NameError) as e:
            # get_type_hints can fail for various reasons:
            # - TypeError: if func is not a callable
            # - AttributeError: if func doesn't have required attributes
            # - NameError: if type hints reference undefined names
            logger.debug(f"Could not extract type hints for {func.__name__}: {e}")
            parameter_types = dict.fromkeys(parameters, 'Any')
            return_type = 'Any'

        # Get module and class information
        module = func.__module__ if hasattr(func, '__module__') else 'unknown'
        class_name = None
        if hasattr(func, '__qualname__') and '.' in func.__qualname__:
            class_name = func.__qualname__.rsplit('.', 1)[0]

        signature_hash = self.compute_signature_hash(func)

        func_sig = FunctionSignature(
            module=module,
            class_name=class_name,
            function_name=func.__name__,
            parameters=parameters,
            parameter_types=parameter_types,
            return_type=return_type,
            signature_hash=signature_hash
        )

        # Store in registry
        key = self._get_function_key(module, class_name, func.__name__)
        self.signatures[key] = func_sig

        return func_sig

    def _get_function_key(self, module: str, class_name: str | None, func_name: str) -> str:
        """Generate a unique key for a function"""
        if class_name:
            return f"{module}.{class_name}.{func_name}"
        return f"{module}.{func_name}"

    def get_signature(self, module: str, class_name: str | None, func_name: str) -> FunctionSignature | None:
        """Retrieve a stored signature"""
        key = self._get_function_key(module, class_name, func_name)
        return self.signatures.get(key)

    def has_signature_changed(self, func: Callable) -> tuple[bool, FunctionSignature | None, FunctionSignature | None]:
        """Check if a function's signature has changed from the registered version"""
        module = func.__module__ if hasattr(func, '__module__') else 'unknown'
        class_name = None
        if hasattr(func, '__qualname__') and '.' in func.__qualname__:
            class_name = func.__qualname__.rsplit('.', 1)[0]

        old_sig = self.get_signature(module, class_name, func.__name__)
        if old_sig is None:
            return False, None, None  # No previous signature

        new_sig = self.register_function(func)
        changed = old_sig.signature_hash != new_sig.signature_hash

        return changed, old_sig, new_sig

    def save(self) -> None:
        """Save registry to disk"""
        self.registry_path.parent.mkdir(parents=True, exist_ok=True)

        registry_data = {
            key: sig.to_dict()
            for key, sig in self.signatures.items()
        }

        with open(self.registry_path, 'w') as f:
            json.dump(registry_data, f, indent=2)

        logger.info(f"Saved {len(self.signatures)} signatures to {self.registry_path}")

    def load(self) -> None:
        """Load registry from disk"""
        if not self.registry_path.exists():
            logger.info(f"No existing registry found at {self.registry_path}")
            return

        try:
            with open(self.registry_path) as f:
                registry_data = json.load(f)

            self.signatures = {
                key: FunctionSignature(**data)
                for key, data in registry_data.items()
            }

            logger.info(f"Loaded {len(self.signatures)} signatures from
{self.registry_path}")
        except Exception as e:
            logger.error(f"Failed to load registry: {e}")

# Global registry instance
_signature_registry = SignatureRegistry()

# ============================================================================
# RUNTIME VALIDATION DECORATOR
# ============================================================================

def validate_signature(enforce: bool = True, track: bool = True):
    """
    Decorator to validate function calls against expected signatures at runtime

    Args:
        enforce: If True, raise TypeError on signature violations
        track: If True, register signature in the global registry

    Example:
        @validate_signature(enforce=True)
        def my_function(arg1: str, arg2: int) -> bool:
            return True
    """
    def decorator(func: F) -> F:
        # Register function signature if tracking is enabled
        if track:
            _signature_registry.register_function(func)

        # Get the function signature
        sig = inspect.signature(func)

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Bind arguments to signature
            try:
                bound_args = sig.bind(*args, **kwargs)
                bound_args.apply_defaults()
            except TypeError as e:
                error_msg = (
                    f"Signature mismatch in {func.__module__}.{func.__qualname__}: 
{e}\n"
                    f"Expected signature: {sig}\n"
                    f"Called with args: {args}, kwargs: {kwargs}"
                )
                logger.error(error_msg)

                if enforce:
                    raise TypeError(error_msg) from e
                else:
                    logger.warning(f"Signature validation failed but enforcement is 
disabled: {e}")

            # Call the original function
            return func(*args, **kwargs)

        return wrapper  # type: ignore

    return decorator

def validate_call_signature(func: Callable, *args, **kwargs) -> bool:
    """
    Validate that a function call matches the expected signature without actually 
calling it

    Args:
        func: Function to validate
        *args: Positional arguments
        **kwargs: Keyword arguments

    Returns:
        True if signature is valid, False otherwise
    """
    try:
        sig = inspect.signature(func)
        sig.bind(*args, **kwargs)
        return True
    except TypeError:
        return False

# ============================================================================
# STATIC SIGNATURE AUDITOR
# ============================================================================

@dataclass
class SignatureMismatch:
    """Represents a detected signature mismatch"""
    caller_module: str
    caller_function: str
    caller_line: int
    callee_module: str
    callee_class: str | None
    callee_function: str
    expected_signature: str
    actual_call: str
    severity: str  # 'high', 'medium', 'low'
    description: str

class SignatureAuditor:
    """
    Static introspection tool to cross-validate function definitions against call sites
    Implements automated signature consistency audit from the mitigation plan
    """

    def __init__(self) -> None:
        self.mismatches: list[SignatureMismatch] = []
        self.call_graph: dict[str, list[tuple[str, int, list[str], dict[str, str]]]] =
{}

    def audit_module(self, module_path: Path) -> list[SignatureMismatch]:
        """
        Audit a Python module for signature mismatches

        Args:
            module_path: Path to the Python module

        Returns:
            List of detected signature mismatches
        """
        logger.info(f"Auditing module: {module_path}")

        # Skip test files, virtual environments, and build directories
        exclude_patterns = ['test', 'venv', '.venv', '__pycache__', '.git', 'build',
'dist']
        if any(module_path.match(f'*/{pattern}/*') or module_path.match(f'*/{pattern}')
               for pattern in exclude_patterns):
            logger.debug(f"Skipping excluded path: {module_path}")
            return []

        try:
            with open(module_path, encoding='utf-8') as f:
                source_code = f.read()

            tree = ast.parse(source_code, filename=str(module_path))

            # Extract function definitions
            function_defs = self._extract_function_definitions(tree, module_path.stem)

            # Extract function calls
            function_calls = self._extract_function_calls(tree, module_path.stem)

            # Cross-validate
            mismatches = self._cross_validate(function_defs, function_calls)

            self.mismatches.extend(mismatches)

            return mismatches
        except Exception as e:
            logger.error(f"Failed to audit {module_path}: {e}")
            return []

        def _extract_function_definitions(self, tree: ast.AST, module_name: str) ->
dict[str, ast.FunctionDef]:
        """Extract all function definitions from AST"""
        functions = {}

        for node in ast.walk(tree):
            if isinstance(node, ast.FunctionDef):
                # Generate full qualified name
                full_name = f"{module_name}.{node.name}"
                functions[full_name] = node

        return functions

        def _extract_function_calls(self, tree: ast.AST, module_name: str) ->
list[tuple[str, int, ast.Call]]:
        """Extract all function calls from AST"""
        calls = []

        for node in ast.walk(tree):
            if isinstance(node, ast.Call):
                # Try to get the function name
                func_name = None
                if isinstance(node.func, ast.Name):
                    func_name = node.func.id
                elif isinstance(node.func, ast.Attribute):
                    func_name = node.func.attr

                if func_name:
                    calls.append((func_name, node.lineno, node))

        return calls

    def _cross_validate(
        self,
        function_defs: dict[str, ast.FunctionDef],
        function_calls: list[tuple[str, int, ast.Call]]
    ) -> list[SignatureMismatch]:
        """Cross-validate function calls against definitions"""
        mismatches = []

        # This is a simplified implementation
        # A full implementation would need more sophisticated analysis

        return mismatches

    def export_report(self, output_path: Path) -> None:
        """Export audit report to JSON"""
        output_path.parent.mkdir(parents=True, exist_ok=True)
        report = {
            "audit_timestamp": datetime.now().isoformat(),
            "total_mismatches": len(self.mismatches),
            "mismatches": [asdict(m) for m in self.mismatches]
        }

        with open(output_path, 'w') as f:
            json.dump(report, f, indent=2)

        logger.info(f"Exported audit report to {output_path}")

# ============================================================================
# COMPATIBILITY LAYER
# ============================================================================

def create_adapter(
    func: Callable,
    old_params: list[str],
    new_params: list[str],
    param_mapping: dict[str, str] | None = None
) -> Callable:
    """
    Create a backward-compatible adapter for a function with changed signature

    Args:
        func: The new function with updated signature
        old_params: List of old parameter names
        new_params: List of new parameter names
        param_mapping: Optional mapping from old to new parameter names

    Returns:
        Adapter function that accepts old signature and calls new function
    """
    param_mapping = param_mapping or {}

    @functools.wraps(func)
    def adapter(*args, **kwargs):
        # Remap old parameter names to new ones
        new_kwargs = {}
        for old_key, value in kwargs.items():
            new_key = param_mapping.get(old_key, old_key)
            new_kwargs[new_key] = value

        return func(*args, **new_kwargs)

    return adapter

# ============================================================================
# MODULE INITIALIZATION
# ============================================================================

def initialize_signature_registry(project_root: Path) -> None:
    """
    Initialize signature registry by scanning all Python files in the project
    Args:
        project_root: Root directory of the project
    """
    logger.info(f"Initializing signature registry for project: {project_root}")

    python_files = list(project_root.glob("**/*.py"))
    logger.info(f"Found {len(python_files)} Python files")

    # This would require dynamic import which is complex
    # For now, we rely on decorators to register functions

    _signature_registry.save()

def audit_project_signatures(project_root: Path, output_path: Path | None = None) -> list[SignatureMismatch]:
    """
    Audit all Python files in a project for signature mismatches

    Args:
        project_root: Root directory of the project
        output_path: Optional path to save audit report

    Returns:
        List of detected signature mismatches
    """
    auditor = SignatureAuditor()

    python_files = list(project_root.glob("**/*.py"))
    logger.info(f"Auditing {len(python_files)} Python files")

    # Define patterns to exclude
    exclude_patterns = ['test', 'venv', '.venv', '__pycache__', '.git', 'build', 'dist']

    all_mismatches = []
    for py_file in python_files:
        # Skip excluded patterns
        if any(py_file.match(f'*/{pattern}/*') or py_file.match(f'*/{pattern}')
               for pattern in exclude_patterns):
            continue

        mismatches = auditor.audit_module(py_file)
        all_mismatches.extend(mismatches)

    if output_path:
        auditor.export_report(output_path)

    logger.info(f"Audit complete: {len(all_mismatches)} mismatches detected")

    return all_mismatches

# ============================================================================
# CLI INTERFACE
# ============================================================================
# Note: Main entry point removed to maintain I/O boundary separation.
# For CLI usage, see examples/ directory or create a dedicated CLI script.
```