src/farfan_pipeline/infrastructure/contractual/dura_lex/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/infrastructure/contractual/dura_lex/determinism_helpers.py

```
"""
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/infrastructure/contractual/dura_lex/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:
```

```
    """
    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/infrastructure/contractual/dura_lex/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/infrastructure/contractual/dura_lex/enhanced_contracts.py

```
"""
Enhanced Contract System with Pydantic - Production Grade
============================================================

Strict contract definitions with cryptographic verification, deterministic
execution guarantees, and comprehensive validation.

Features:
- Static typing with Pydantic BaseModel
- Schema versioning for backward compatibility
- Cryptographic content digests (SHA-256)
- UTC timestamps (ISO-8601)
- Domain-specific exceptions
- Structured JSON logging
- Flow compatibility validation

Author: Policy Analytics Research Unit
Version: 2.0.0
License: Proprietary
"""

from __future__ import annotations

import hashlib
import json
import logging
import uuid
from datetime import datetime, timezone
from typing import Any

from pydantic import BaseModel, ConfigDict, Field, field_validator
from farfan_pipeline.core.parameters import ParameterLoaderV2

# ============================================================================
# DOMAIN-SPECIFIC EXCEPTIONS
# ============================================================================

class ContractValidationError(Exception):
    """Raised when contract validation fails."""

     def __init__(self, message: str, field: str | None = None, event_id: str | None =
None) -> None:
        self.field = field
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class DataIntegrityError(Exception):
    """Raised when data integrity checks fail (e.g., hash mismatch)."""

     def __init__(self, message: str, expected: str | None = None, got: str | None =
None, event_id: str | None = None) -> None:
```

```
        self.expected = expected
        self.got = got
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class SystemConfigError(Exception):
    """Raised when system configuration is invalid."""

    def __init__(self, message: str, config_key: str | None = None, event_id: str | None
= None) -> None:
        self.config_key = config_key
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


class FlowCompatibilityError(Exception):
    """Raised when data flow between components is incompatible."""

    def __init__(self, message: str, producer: str | None = None, consumer: str | None =
None, event_id: str | None = None) -> None:
        self.producer = producer
        self.consumer = consumer
        self.event_id = event_id or str(uuid.uuid4())
        super().__init__(f"[{self.event_id}] {message}")


# ============================================================================
# UTILITY FUNCTIONS FOR DETERMINISM AND VALIDATION
# ============================================================================

def compute_content_digest(content: str | bytes | dict[str, Any]) -> str:
    """
    Compute SHA-256 digest of content in a deterministic way.

    Args:
        content: String, bytes, or dict to hash

    Returns:
        Hexadecimal SHA-256 digest

    Examples:
        >>> digest = compute_content_digest("test")
        >>> len(digest)
        64
        >>> digest == compute_content_digest("test")  # Deterministic
        True
    """
    if isinstance(content, dict):
        # Sort keys for deterministic JSON
        content_str = json.dumps(content, sort_keys=True, ensure_ascii=True)
        content_bytes = content_str.encode('utf-8')
    elif isinstance(content, str):
        content_bytes = content.encode('utf-8')
```

```
    elif isinstance(content, bytes):
        content_bytes = content
    else:
        raise ContractValidationError(
            f"Cannot compute digest for type {type(content).__name__}",
            field="content"
        )

    return hashlib.sha256(content_bytes).hexdigest()


def utc_now_iso() -> str:
    """
    Get current UTC timestamp in ISO-8601 format.

    Returns:
        ISO-8601 timestamp string (UTC timezone)

    Examples:
        >>> ts = utc_now_iso()
        >>> 'T' in ts and 'Z' in ts
        True
    """
    return datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')


# ============================================================================
# BASE CONTRACT MODEL
# ============================================================================

class BaseContract(BaseModel):
    """
    Base contract model with common fields for all contracts.

    All contracts must include:
    - schema_version: Semantic version for contract evolution
    - timestamp_utc: ISO-8601 UTC timestamp
    - correlation_id: UUID for request tracing
    """

    model_config = ConfigDict(
        frozen=True,  # Immutable for safety
        extra='forbid',  # Reject unknown fields
        validate_assignment=True,
        str_strip_whitespace=True,
    )

    schema_version: str = Field(
        default="2.0.0",
        description="Contract schema version (semantic versioning)",
        pattern=r"^\d+\.\d+\.\d+$"
    )

    timestamp_utc: str = Field(
```

```
        default_factory=utc_now_iso,
        description="UTC timestamp in ISO-8601 format"
    )

    correlation_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="UUID for request correlation and tracing"
    )

    @field_validator('timestamp_utc')
    @classmethod
    def validate_timestamp(cls, v: str) -> str:
        """Validate timestamp is ISO-8601 format and UTC."""
        try:
            dt = datetime.fromisoformat(v.replace('Z', '+00:00'))
            # Ensure UTC
            if dt.tzinfo is None or dt.utcoffset() != timezone.utc.utcoffset(None):
                raise ValueError("Timestamp must be UTC")
            return v
        except (ValueError, AttributeError) as e:
            raise ContractValidationError(
                f"Invalid ISO-8601 timestamp: {v}",
                field="timestamp_utc"
            ) from e


# ============================================================================
# DOCUMENT CONTRACTS - V2
# ============================================================================

class DocumentMetadataV2(BaseContract):
    """
    Enhanced document metadata with cryptographic verification.

    Attributes:
        file_path: Absolute path to document
        file_name: Document filename
        num_pages: Number of pages
        file_size_bytes: File size in bytes
        content_digest: SHA-256 hash of file content
        policy_unit_id: Unique identifier for policy unit
        encoding: Character encoding (default: utf-8)
    """

    file_path: str = Field(..., description="Absolute path to document")
    file_name: str = Field(..., description="Document filename", min_length=1)
    num_pages: int = Field(..., description="Number of pages", ge=1)
    file_size_bytes: int = Field(..., description="File size in bytes", ge=0)
         content_digest:  str  =  Field(...,  description="SHA-256  hash  of  content",
pattern=r"^[a-f0-9]{64}$")
    policy_unit_id: str = Field(..., description="Unique policy unit identifier")
    encoding: str = Field(default="utf-8", description="Character encoding")

    # Optional metadata
```

```
    pdf_metadata: dict[str, Any] | None = Field(default=None, description="PDF metadata dictionary")
    author: str | None = Field(default=None, description="Document author")
    title: str | None = Field(default=None, description="Document title")
    creation_date: str | None = Field(default=None, description="Document creation date")


class ProcessedTextV2(BaseContract):
    """
    Enhanced processed text with input/output validation.

    Attributes:
        raw_text: Original unprocessed text
        normalized_text: Normalized/cleaned text
        language: Detected language code
        input_digest: SHA-256 of raw_text input
        output_digest: SHA-256 of normalized_text output
        policy_unit_id: Policy unit identifier
        processing_latency_ms: Processing time in milliseconds
    """

    raw_text: str = Field(..., description="Original unprocessed text", min_length=1)
    normalized_text: str = Field(..., description="Normalized/cleaned text", min_length=1)
    language: str = Field(..., description="ISO 639-1 language code", pattern=r"^[a-z]{2}$")
    input_digest: str = Field(..., description="SHA-256 of raw_text", pattern=r"^[a-f0-9]{64}$")
    output_digest: str = Field(..., description="SHA-256 of normalized_text", pattern=r"^[a-f0-9]{64}$")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
    processing_latency_ms: float = Field(..., description="Processing latency in ms", ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__init__", "auto_param_L236_89", 0.0))

    # Optional fields
    sentences: list[str] | None = Field(default=None, description="Sentence segmentation")
    sections: list[dict[str, Any]] | None = Field(default=None, description="Document sections")
    payload_size_bytes: int | None = Field(default=None, description="Payload size", ge=0)

    @field_validator('input_digest')
    @classmethod
    def validate_input_digest(cls, v: str, info) -> str:
        """Verify input digest matches raw_text if available."""
        # This is validated post-construction
        return v


# ============================================================================
# ANALYSIS CONTRACTS - V2
```

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

class AnalysisInputV2(BaseContract):
    """
    Enhanced analysis input with cryptographic verification.

    Attributes:
        text: Input text to analyze
        document_id: Unique document identifier
        policy_unit_id: Policy unit identifier
        input_digest: SHA-256 of input text
        payload_size_bytes: Size of input payload
    """

    text: str = Field(..., description="Input text to analyze", min_length=1)
    document_id: str = Field(..., description="Unique document identifier")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
    input_digest: str = Field(
        ...,
        description="SHA-256 hash of input text",
        pattern=r"^[a-f0-9]{64}$"
    )
    payload_size_bytes: int = Field(..., description="Payload size in bytes", ge=0)

    # Optional context
       metadata: dict[str, Any] | None = Field(default=None, description="Additional
metadata")
       context: dict[str, Any] | None = Field(default=None, description="Execution
context")
       sentences: list[str] | None = Field(default=None, description="Pre-segmented
sentences")

    @classmethod
    def create_from_text(
        cls,
        text: str,
        document_id: str,
        policy_unit_id: str,
        **kwargs: Any
    ) -> AnalysisInputV2:
        """
        Factory method to create AnalysisInputV2 with auto-computed digest.

        Args:
            text: Input text
            document_id: Document ID
            policy_unit_id: Policy unit ID
            **kwargs: Additional optional fields

        Returns:
            Validated AnalysisInputV2 instance
        """
        input_digest = compute_content_digest(text)
        payload_size_bytes = len(text.encode('utf-8'))
```

```
        return cls(
            text=text,
            document_id=document_id,
            policy_unit_id=policy_unit_id,
            input_digest=input_digest,
            payload_size_bytes=payload_size_bytes,
            **kwargs
        )


class AnalysisOutputV2(BaseContract):
    """
    Enhanced analysis output with confidence bounds and validation.

    Attributes:
        dimension: Analysis dimension
        category: Result category
                                                confidence:      Confidence      score
[ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__in
it__",                              "auto_param_L322_38",                                0.0),
ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__ini
t__", "auto_param_L322_43", 1.0)]
        matches: Evidence matches
        output_digest: SHA-256 of output content
        policy_unit_id: Policy unit identifier
        processing_latency_ms: Processing time in milliseconds
    """

    dimension: str = Field(..., description="Analysis dimension", min_length=1)
    category: str = Field(..., description="Result category", min_length=1)
               confidence:    float    =    Field(...,    description="Confidence    score",
ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__
init__",                             "auto_param_L331_70",                                0.0),
le=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__
init__", "auto_param_L331_78", 1.0))
    matches: list[str] = Field(..., description="Evidence matches")
             output_digest:   str   =   Field(...,   description="SHA-256   of   output",
pattern=r"^[a-f0-9]{64}$")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
      processing_latency_ms: float = Field(..., description="Processing latency in ms",
ge=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.FlowCompatibilityError.__
init__", "auto_param_L335_89", 0.0))

    # Optional fields
    positions: list[int] | None = Field(default=None, description="Match positions")
    evidence: list[str] | None = Field(default=None, description="Supporting evidence")
    warnings: list[str] | None = Field(default=None, description="Validation warnings")
      payload_size_bytes:  int  |  None  =  Field(default=None,  description="Output  payload
size", ge=0)

    @field_validator('confidence')
    @classmethod
    def validate_confidence_numerical_stability(cls, v: float) -> float:
```

```
        """Ensure confidence is numerically stable and within bounds."""
        if not (0.0 <= v <= 1.0):
            raise ContractValidationError(
                f"Confidence must be in [0.0, 1.0], got {v}",
                field="confidence"
            )
        # Round to avoid floating point precision issues
        return round(v, 6)


# ============================================================================
# EXECUTION CONTRACTS - V2
# ============================================================================

class ExecutionContextV2(BaseContract):
    """
    Enhanced execution context with full observability.

    Attributes:
        class_name: Executor class name
        method_name: Method being executed
        document_id: Document identifier
        policy_unit_id: Policy unit identifier
        execution_id: Unique execution identifier
        parent_correlation_id: Parent request correlation ID
    """

    class_name: str = Field(..., description="Executor class name", min_length=1)
    method_name: str = Field(..., description="Method being executed", min_length=1)
    document_id: str = Field(..., description="Document identifier")
    policy_unit_id: str = Field(..., description="Policy unit identifier")
    execution_id: str = Field(
        default_factory=lambda: str(uuid.uuid4()),
        description="Unique execution identifier"
    )
    parent_correlation_id: str | None = Field(
        default=None,
        description="Parent correlation ID for nested calls"
    )

    # Optional context
    raw_text: str | None = Field(default=None, description="Raw input text")
    text: str | None = Field(default=None, description="Processed text")
    metadata: dict[str, Any] | None = Field(default=None, description="Metadata")
    tables: dict[str, Any] | None = Field(default=None, description="Extracted tables")
    sentences: list[str] | None = Field(default=None, description="Sentences")


# ============================================================================
# STRUCTURED LOGGING HELPER
# ============================================================================

class StructuredLogger:
    """
```

```
Structured JSON logger for observability.

Logs include:
- correlation_id for tracing
- latencies per operation
- payload sizes
- cryptographic fingerprints
- NO PII
"""

def __init__(self, name: str) -> None:
    """Initialize logger with name."""
    self.logger = logging.getLogger(name)
    self.logger.setLevel(logging.INFO)

def log_contract_validation(
    self,
    contract_type: str,
    correlation_id: str,
    success: bool,
    latency_ms: float,
    payload_size_bytes: int = 0,
    content_digest: str | None = None,
    error: str | None = None
) -> None:
    """Log contract validation event."""
    log_entry = {
        "event": "contract_validation",
        "contract_type": contract_type,
        "correlation_id": correlation_id,
        "success": success,
        "latency_ms": round(latency_ms, 3),
        "payload_size_bytes": payload_size_bytes,
        "timestamp_utc": utc_now_iso(),
    }

    if content_digest:
        log_entry["content_digest"] = content_digest

    if error:
        log_entry["error"] = error

    self.logger.info(json.dumps(log_entry, sort_keys=True))

def log_execution(
    self,
    operation: str,
    correlation_id: str,
    success: bool,
    latency_ms: float,
    **kwargs: Any
) -> None:
    """Log execution event with additional context."""
    log_entry = {
```

```
            "event": "execution",
            "operation": operation,
            "correlation_id": correlation_id,
            "success": success,
            "latency_ms": round(latency_ms, 3),
            "timestamp_utc": utc_now_iso(),
        }
        log_entry.update(kwargs)

        self.logger.info(json.dumps(log_entry, sort_keys=True))


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

if __name__ == "__main__":
    import doctest

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

    # Contract validation examples
    print("\n" + "="*60)
    print("Contract Validation Examples")
    print("="*60)

    # Example 1: Document metadata
    print("\n1. DocumentMetadataV2 validation:")
    doc_meta = DocumentMetadataV2(
        file_path="/path/to/document.pdf",
        file_name="document.pdf",
        num_pages=10,
        file_size_bytes=1024000,
        content_digest="a" * 64,  # Valid SHA-256 hex
        policy_unit_id="PDM-001"
    )
    print(f"   ? Valid: correlation_id={doc_meta.correlation_id[:8]}...")

    # Example 2: Analysis input with auto-digest
    print("\n2. AnalysisInputV2 with auto-computed digest:")
    analysis_input = AnalysisInputV2.create_from_text(
        text="Sample policy text for analysis",
        document_id="DOC-123",
        policy_unit_id="PDM-001"
    )
    print(f"   ? Valid: input_digest={analysis_input.input_digest[:16]}...")
    print(f"   ? Payload size: {analysis_input.payload_size_bytes} bytes")

    # Example 3: Analysis output with confidence validation
    print("\n3. AnalysisOutputV2 with confidence bounds:")
    analysis_output = AnalysisOutputV2(
        dimension="Dimension1",
```

```
        category="CategoryA",

confidence=ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.StructuredLogger.
__init__",      "auto_param_L509_19",      0.85),            #      Must      be      in
[ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.StructuredLogger.__init__",
"auto_param_L509_40",                                                              0.0),
ParameterLoaderV2.get("farfan_core.utils.enhanced_contracts.StructuredLogger.__init__",
"auto_param_L509_45", 1.0)]
        matches=["evidence1", "evidence2"],
        output_digest="b" * 64,
        policy_unit_id="PDM-001",
        processing_latency_ms=123.456
    )
    print(f"   ? Valid: confidence={analysis_output.confidence}")

    # Example 4: Structured logging
    print("\n4. Structured logging example:")
    logger = StructuredLogger("test_logger")
    logger.log_contract_validation(
        contract_type="AnalysisInputV2",
        correlation_id=analysis_input.correlation_id,
        success=True,
        latency_ms=5.2,
        payload_size_bytes=analysis_input.payload_size_bytes,
        content_digest=analysis_input.input_digest
    )
    print("   ? JSON log emitted to logger")

    # Example 5: Exception handling
    print("\n5. Domain-specific exceptions:")
    try:
        raise ContractValidationError("Invalid field", field="test_field")
    except ContractValidationError as e:
        print(f"   ? ContractValidationError: {e}")
        print(f"   ? Event ID: {e.event_id}")

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

src/farfan_pipeline/infrastructure/contractual/dura_lex/failure_fallback.py

```
"""
Failure & Fallback Contract (FFC) - Implementation
"""
from typing import Callable, Any, Dict, Type, Tuple

class FailureFallbackContract:
    @staticmethod
    def execute_with_fallback(
        func: Callable,
        fallback_value: Any,
        expected_exceptions: Tuple[Type[Exception], ...]
    ) -> Any:
        """
        Executes func. If it raises an expected exception, returns fallback_value.
        Ensures determinism and no side effects (simulated).
        """
        try:
            return func()
        except expected_exceptions:
            return fallback_value

    @staticmethod
    def verify_fallback_determinism(
        func: Callable,
        fallback_value: Any,
        exception_type: Type[Exception]
    ) -> bool:
        """
        Verifies that repeated failures produce identical fallback values.
        """
             res1 = FailureFallbackContract.execute_with_fallback(func, fallback_value,
(exception_type,))
             res2 = FailureFallbackContract.execute_with_fallback(func, fallback_value,
(exception_type,))
        return res1 == res2
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/governance.py

```
"""
Contract Governance Utilities
"""
from typing import Type, Any, Callable
import functools

def uses_contract(contract_class: Type[Any]) -> Callable:
    """
      Decorator to explicitly declare that a function or class relies on a specific contract.
        This serves as documentation and allows for static analysis of contract dependencies.

    Usage:
        @uses_contract(RoutingContract)
        def my_function():
            ...
    """
    def decorator(obj: Any) -> Any:
        if not hasattr(obj, "_contract_dependencies"):
            obj._contract_dependencies = []
        obj._contract_dependencies.append(contract_class)

        # If it's a function, wrap it to preserve metadata
        if callable(obj) and not isinstance(obj, type):
            @functools.wraps(obj)
            def wrapper(*args, **kwargs):
                return obj(*args, **kwargs)
            wrapper._contract_dependencies = obj._contract_dependencies
            return wrapper

        return obj
    return decorator
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/idempotency_dedup.py

```
"""
Idempotency & De-dup Contract (IDC) - Implementation
"""
import hashlib
import json
from typing import List, Dict, Any, Set

class EvidenceStore:
    def __init__(self):
        self.evidence: Dict[str, Any] = {} # content_hash -> evidence
        self.duplicates_blocked = 0

    def add(self, item: Dict[str, Any]):
        # Calculate content hash
                                    content_hash    =    hashlib.blake2b(json.dumps(item,
sort_keys=True).encode()).hexdigest()

        if content_hash in self.evidence:
            self.duplicates_blocked += 1
        else:
            self.evidence[content_hash] = item

    def state_hash(self) -> str:
        # Hash of sorted keys to ensure order independence
        sorted_keys = sorted(self.evidence.keys())
        return hashlib.blake2b(json.dumps(sorted_keys).encode()).hexdigest()

class IdempotencyContract:
    @staticmethod
    def verify_idempotency(items: List[Dict[str, Any]]) -> Dict[str, Any]:
        store = EvidenceStore()
        for item in items:
            store.add(item)

        return {
            "state_hash": store.state_hash(),
            "duplicates_blocked": store.duplicates_blocked,
            "count": len(store.evidence)
        }
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/json_contract_loader.py

```
"""Utility helpers to load and validate JSON contract documents."""
from __future__ import annotations

import hashlib
import json
from dataclasses import dataclass
from pathlib import Path
from typing import TYPE_CHECKING, Union

if TYPE_CHECKING:
    from collections.abc import Iterable, Mapping

PathLike = Union[str, Path]

def _canonical_dump(payload: Mapping[str, object]) -> str:
       return  json.dumps(payload,  ensure_ascii=False,  sort_keys=True,  separators=(",",
":"))

@dataclass(frozen=True)
class ContractDocument:
    """Materialized JSON contract with checksum information."""

    path: Path
    payload: dict[str, object]
    checksum: str

@dataclass
class ContractLoadReport:
    """Result of attempting to load multiple contract documents."""

    documents: dict[str, ContractDocument]
    errors: list[str]

    @property
    def is_successful(self) -> bool:
        return not self.errors

    def summary(self) -> str:
        parts = [f"contracts={len(self.documents)}"]
        if self.errors:
            parts.append(f"errors={len(self.errors)}")
        return ", ".join(parts)

class JSONContractLoader:
    """Load JSON contract files and compute integrity metadata.

    ARCHITECTURAL BOUNDARY: This loader is for generic JSON contracts ONLY.
    It must NOT be used to load questionnaire_monolith.json directly.

    For questionnaire access, use:
                -    factory.load_questionnaire()    for    canonical    loading    (returns
CanonicalQuestionnaire)
```

```
    - QuestionnaireResourceProvider for pattern extraction
    """

    def __init__(self, base_path: Path | None = None) -> None:
        self.base_path = base_path or Path(__file__).resolve().parent

    def load(self, paths: Iterable[PathLike]) -> ContractLoadReport:
        documents: dict[str, ContractDocument] = {}
        errors: list[str] = []
        for raw in paths:
            path = self._resolve_path(raw)
            try:
                payload = self._read_payload(path)
            except (FileNotFoundError, json.JSONDecodeError, ValueError) as exc:
                errors.append(f"{path}: {exc}")
                continue

                                                                    checksum      =
hashlib.sha256(_canonical_dump(payload).encode("utf-8")).hexdigest()
                    documents[str(path)]  =  ContractDocument(path=path,  payload=payload,
checksum=checksum)
        return ContractLoadReport(documents=documents, errors=errors)

     def load_directory(self, relative_directory: PathLike, pattern: str = "*.json") ->
ContractLoadReport:
        directory = self._resolve_path(relative_directory)
        if not directory.exists():
                 return ContractLoadReport(documents={}, errors=[f"Directory not found:
{directory}"])
        if not directory.is_dir():
                    return  ContractLoadReport(documents={},  errors=[f"Not  a  directory:
{directory}"])

        paths = sorted(directory.glob(pattern))
        return self.load(paths)

    # ------------------------------------------------------------------
    # Helpers
    # ------------------------------------------------------------------
    def _resolve_path(self, raw: PathLike) -> Path:
        path = Path(raw)
        if not path.is_absolute():
            path = self.base_path / path
        return path

    @staticmethod
    def _read_payload(path: Path) -> dict[str, object]:
        # ARCHITECTURAL GUARD: Block unauthorized questionnaire monolith access
        if path.name == "questionnaire_monolith.json":
            raise ValueError(
                "ARCHITECTURAL VIOLATION: questionnaire_monolith.json must ONLY be "
                          "loaded  via  factory.load_questionnaire()  which  enforces  hash
verification. "
                "Use factory.load_questionnaire() for canonical loading."
```

```
            )

        text = path.read_text(encoding="utf-8")
        data = json.loads(text)
        if not isinstance(data, dict):
            raise ValueError("Contract document must be a JSON object")
        return data

__all__ = [
    "ContractDocument",
    "ContractLoadReport",
    "JSONContractLoader",
]
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/monotone_compliance.py

```
"""
Monotone Compliance Contract (MCC) - Implementation
"""
from typing import Set, Dict, Any
from enum import IntEnum

class Label(IntEnum):
    UNSAT = 0
    PARTIAL = 1
    SAT = 2

class MonotoneComplianceContract:
    @staticmethod
    def evaluate(evidence: Set[str], rules: Dict[str, Any]) -> Label:
        """
        Evaluates label based on evidence and Horn-like clauses.
        Simple logic:
        - SAT if all 'sat_reqs' present
        - PARTIAL if all 'partial_reqs' present
        - UNSAT otherwise
        """
        sat_reqs = set(rules.get("sat_reqs", []))
        partial_reqs = set(rules.get("partial_reqs", []))

        if sat_reqs.issubset(evidence):
            return Label.SAT
        elif partial_reqs.issubset(evidence):
            return Label.PARTIAL
        else:
            return Label.UNSAT

    @staticmethod
    def verify_monotonicity(
        evidence_subset: Set[str],
        evidence_superset: Set[str],
        rules: Dict[str, Any]
    ) -> bool:
        """
        Verifies label(E') >= label(E) for E ? E'.
        """
        if not evidence_subset.issubset(evidence_superset):
            raise ValueError("Subset is not contained in superset")

        l1 = MonotoneComplianceContract.evaluate(evidence_subset, rules)
        l2 = MonotoneComplianceContract.evaluate(evidence_superset, rules)

        return l2 >= l1
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/permutation_invariance.py

```
"""
Permutation-Invariance Contract (PIC) - Implementation
"""
import hashlib
from typing import List, Any, Callable

class PermutationInvarianceContract:
    @staticmethod
    def aggregate(items: List[Any], transform: Callable[[Any], float]) -> float:
        """
        Implements f(S) = ?(? ?(x)) pattern for permutation invariance.
        Here, sum is the aggregation function (symmetric).
        """
        # ?(x) = transform(x)
        transformed = [transform(x) for x in items]

        # ? ?(x) - Sum is order-independent (within floating point limits, usually)
         # For strict bitwise invariance with floats, we might need to sort or use exact
arithmetic.
        # But the requirement asks for "numerical tolerance".
        total = sum(transformed)

        # ?(x) = identity (for this example)
        return total

    @staticmethod
    def verify_invariance(items: List[Any], transform: Callable[[Any], float]) -> str:
        """
        Calculates digest of the aggregation.
        """
        result = PermutationInvarianceContract.aggregate(items, transform)
        return hashlib.blake2b(str(result).encode()).hexdigest()
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/phase_zero_contracts.py

```
"""
CONTRACT DEFINITIONS - Frozen Data Shapes
==========================================

TypedDict and Protocol definitions for API contracts across modules.
All data shapes must be versioned and adapters maintained for one release cycle.

Purpose: Replace ad-hoc dicts with typed structures to prevent:
- unexpected keyword argument errors
- missing required positional arguments
- 'str' object has no attribute 'text' errors
- 'bool' object is not iterable
- unhashable type: 'dict' in sets

Version 2.0 Enhancement:
- Pydantic-based contracts with strict validation (enhanced_contracts.py)
- Backward compatibility with V1 TypedDict contracts maintained
- Domain-specific exceptions and structured logging
- Cryptographic content verification and deterministic execution
"""

from __future__ import annotations

from dataclasses import dataclass
from typing import (
    TYPE_CHECKING,
    Any,
    Literal,
    Protocol,
    TypedDict,
)

if TYPE_CHECKING:
    from collections.abc import Iterable, Mapping, Sequence
    from pathlib import Path

# ============================================================================
# V2 ENHANCED CONTRACTS - Pydantic-based with strict validation
# ============================================================================
# Import V2 contracts from enhanced_contracts module
# Use these for new code; V1 contracts maintained for backward compatibility
from farfan_pipeline.utils.enhanced_contracts import (
    # Pydantic Models
    AnalysisInputV2,
    AnalysisOutputV2,
    BaseContract,
    # Exceptions
    ContractValidationError,
    DataIntegrityError,
    DocumentMetadataV2,
    ExecutionContextV2,
    FlowCompatibilityError,
```

```
    ProcessedTextV2,
    # Utilities
    StructuredLogger,
    SystemConfigError,
    compute_content_digest,
    utc_now_iso,
)

# ============================================================================
# DOCUMENT CONTRACTS - V1
# ============================================================================

class DocumentMetadataV1(TypedDict, total=True):
    """Document metadata shape - all fields required."""
    file_path: str
    file_name: str
    num_pages: int
    file_size_bytes: int
    file_hash: str

class DocumentMetadataV1Optional(TypedDict, total=False):
    """Optional document metadata fields."""
    pdf_metadata: dict[str, Any]
    author: str
    title: str
    creation_date: str

class ProcessedTextV1(TypedDict, total=True):
    """Shape for processed text output."""
    raw_text: str
    normalized_text: str
    language: str
    encoding: str

class ProcessedTextV1Optional(TypedDict, total=False):
    """Optional processed text fields."""
    sentences: list[str]
    sections: list[dict[str, Any]]
    tables: Mapping[str, Any]

# ============================================================================
# ANALYSIS CONTRACTS - V1
# ============================================================================

class AnalysisInputV1(TypedDict, total=True):
    """Required fields for analysis input - keyword-only."""
    text: str
    document_id: str

class AnalysisInputV1Optional(TypedDict, total=False):
    """Optional fields for analysis input."""
    metadata: Mapping[str, Any]
    context: Mapping[str, Any]
    sentences: Sequence[str]
```

```
class AnalysisOutputV1(TypedDict, total=True):
    """Shape for analysis output."""
    dimension: str
    category: str
    confidence: float
    matches: Sequence[str]

class AnalysisOutputV1Optional(TypedDict, total=False):
    """Optional analysis output fields."""
    positions: Sequence[int]
    evidence: Sequence[str]
    warnings: Sequence[str]

# ============================================================================
# EXECUTION CONTRACTS - V1
# ============================================================================

class ExecutionContextV1(TypedDict, total=True):
    """Execution context for method invocation."""
    class_name: str
    method_name: str
    document_id: str

class ExecutionContextV1Optional(TypedDict, total=False):
    """Optional execution context fields."""
    raw_text: str
    text: str
    metadata: Mapping[str, Any]
    tables: Mapping[str, Any]
    sentences: Sequence[str]

# ============================================================================
# ERROR REPORTING CONTRACTS
# ============================================================================

class ContractMismatchError(TypedDict, total=True):
    """Standard error shape for contract mismatches."""
    error_code: Literal["ERR_CONTRACT_MISMATCH"]
    stage: str
    function: str
    parameter: str
    expected_type: str
    got_type: str
    producer: str
    consumer: str

# ============================================================================
# PROTOCOLS FOR PLUGGABLE BEHAVIOR
# ============================================================================

class TextProcessorProtocol(Protocol):
    """Protocol for text processing components."""
```

```
@calibrated_method("farfan_core.utils.contracts.TextProcessorProtocol.normalize_unicode"
)
    def normalize_unicode(self, text: str) -> str:
        """Normalize unicode characters in text."""
        ...


@calibrated_method("farfan_core.utils.contracts.TextProcessorProtocol.segment_into_sente
nces")
    def segment_into_sentences(self, text: str) -> Sequence[str]:
        """Segment text into sentences."""
        ...

class DocumentLoaderProtocol(Protocol):
    """Protocol for document loading components."""

    @calibrated_method("farfan_core.utils.contracts.DocumentLoaderProtocol.load_pdf")
    def load_pdf(self, *, pdf_path: Path) -> DocumentMetadataV1:
        """Load PDF and return metadata - keyword-only params."""
        ...


@calibrated_method("farfan_core.utils.contracts.DocumentLoaderProtocol.validate_pdf")
    def validate_pdf(self, *, pdf_path: Path) -> bool:
        """Validate PDF file - keyword-only params."""
        ...

class AnalyzerProtocol(Protocol):
    """Protocol for analysis components."""

    def analyze(
        self,
        *,
        text: str,
        document_id: str,
        metadata: Mapping[str, Any] | None = None,
    ) -> AnalysisOutputV1:
        """Analyze text and return structured output - keyword-only params."""
        ...

# ============================================================================
# VALUE OBJECTS (prevent .text on strings)
# ============================================================================

@dataclass(frozen=True, slots=True)
class TextDocument:
    """Wrapper to prevent passing plain str where structured text is required."""
    text: str
    document_id: str
    metadata: Mapping[str, Any]

    def __post_init__(self) -> None:
        """Validate that text is non-empty."""
```

```
        if not isinstance(self.text, str):
            raise TypeError(
                                    f"ERR_CONTRACT_MISMATCH:  text  must  be  str,  got
{type(self.text).__name__}"
            )
        if not self.text:
            raise ValueError("ERR_CONTRACT_MISMATCH: text cannot be empty")

@dataclass(frozen=True, slots=True)
class SentenceCollection:
    """Type-safe collection of sentences (prevents iteration bugs)."""
    sentences: tuple[str, ...]  # Immutable and hashable

    def __post_init__(self) -> None:
        """Validate sentences are strings."""
        if not all(isinstance(s, str) for s in self.sentences):
            raise TypeError(
                "ERR_CONTRACT_MISMATCH: All sentences must be strings"
            )

    def __iter__(self) -> Iterable[str]:
        """Make iterable."""
        return iter(self.sentences)

    def __len__(self) -> int:
        """Return count."""
        return len(self.sentences)

# ============================================================================
# SENTINEL VALUES (avoid None ambiguity)
# ============================================================================

class _MissingSentinel:
    """Sentinel type for missing optional parameters."""

    def __repr__(self) -> str:
        return "<MISSING>"

MISSING: _MissingSentinel = _MissingSentinel()

# ============================================================================
# RUNTIME VALIDATION HELPERS
# ============================================================================

def validate_contract(
    value: Any,
    expected_type: type,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value matches expected contract at runtime.
```

```
    Raises TypeError with structured error message on mismatch.
    """
    if not isinstance(value, expected_type):
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected={expected_type.__name__}, "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise TypeError(error_msg)

def validate_mapping_keys(
    mapping: Mapping[str, Any],
    required_keys: Sequence[str],
    *,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate mapping contains required keys.

    Raises KeyError with structured message on missing keys.
    """
    missing = [key for key in required_keys if key not in mapping]
    if missing:
        error_msg = (
            f"ERR_CONTRACT_MISMATCH["
            f"missing_keys={missing}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )
        raise KeyError(error_msg)

def ensure_iterable_not_string(
    value: Any,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value is iterable but NOT a string or bytes.

    Prevents "'bool' object is not iterable" and "iterate string as tokens" bugs.
    """
    if isinstance(value, (str, bytes)):
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
```

```
            f"expected=Iterable (not str/bytes), "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        )

    try:
        iter(value)
    except TypeError as e:
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected=Iterable, "
            f"got={type(value).__name__}, "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        ) from e

def ensure_hashable(
    value: Any,
    *,
    parameter: str,
    producer: str,
    consumer: str,
) -> None:
    """
    Validate value is hashable (can be added to set or used as dict key).

    Prevents "unhashable type: 'dict'" errors.
    """
    try:
        hash(value)
    except TypeError as e:
        raise TypeError(
            f"ERR_CONTRACT_MISMATCH["
            f"param='{parameter}', "
            f"expected=Hashable, "
            f"got={type(value).__name__} (unhashable), "
            f"producer={producer}, "
            f"consumer={consumer}"
            f"]"
        ) from e


# ============================================================================
# MODULE EXPORTS
# ============================================================================

__all__ = [
    # V2 Enhanced Contracts (Pydantic-based) - RECOMMENDED FOR NEW CODE
    "AnalysisInputV2",
    "AnalysisOutputV2",
```

```
    "BaseContract",
    "DocumentMetadataV2",
    "ExecutionContextV2",
    "ProcessedTextV2",
    # V2 Exceptions
    "ContractValidationError",
    "DataIntegrityError",
    "FlowCompatibilityError",
    "SystemConfigError",
    # V2 Utilities
    "StructuredLogger",
    "compute_content_digest",
    "utc_now_iso",
    # V1 Contracts (TypedDict-based) - BACKWARD COMPATIBILITY
    "AnalysisInputV1",
    "AnalysisInputV1Optional",
    "AnalysisOutputV1",
    "AnalysisOutputV1Optional",
    "AnalyzerProtocol",
    "ContractMismatchError",
    "DocumentLoaderProtocol",
    "DocumentMetadataV1",
    "DocumentMetadataV1Optional",
    "ExecutionContextV1",
    "ExecutionContextV1Optional",
    "MISSING",
    "ProcessedTextV1",
    "ProcessedTextV1Optional",
    "SentenceCollection",
    "TextDocument",
    "TextProcessorProtocol",
    "ensure_hashable",
    "ensure_iterable_not_string",
    "validate_contract",
    "validate_mapping_keys",
]
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/refusal.py

```
"""
Refusal Contract (RefC) - Implementation
"""
from typing import Dict, Any

class RefusalError(Exception):
    pass

class RefusalContract:
    @staticmethod
    def check_prerequisites(context: Dict[str, Any]):
        """
        Confirma que ante prerequisitos fallidos el sistema rehúsa con motivo tipado.
        """
        if "mandatory" not in context:
            raise RefusalError("Missing mandatory field")

        if context.get("alpha", 1.0) > 0.5:
             raise RefusalError("Alpha violation")

        if "sigma" not in context:
            raise RefusalError("Sigma absent")

    @staticmethod
    def verify_refusal(context: Dict[str, Any]) -> str:
        try:
            RefusalContract.check_prerequisites(context)
            return "OK"
        except RefusalError as e:
            return str(e)
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/retriever_contract.py

```
"""
Retriever Contract (ReC) - Implementation
"""
import hashlib
import json
from typing import List, Dict, Any

class RetrieverContract:
    @staticmethod
     def retrieve(query: str, filters: Dict[str, Any], index_hash: str, top_k: int = 5)
-> List[Dict[str, Any]]:
        """
        Simulates hybrid retrieval (patterns+dimension+?+?).
        In a real system, this would call FAISS/Pyserini.
        Here we simulate deterministic retrieval based on inputs.
        """
        # Deterministic simulation
        input_data = f"{query}:{json.dumps(filters, sort_keys=True)}:{index_hash}"
        hasher = hashlib.blake2b(input_data.encode(), digest_size=32)

        results = []
        current_hash = hasher.hexdigest()

        for i in range(top_k):
            doc_hash = hashlib.blake2b(f"{current_hash}:{i}".encode()).hexdigest()
            results.append({
                "id": f"doc_{doc_hash[:8]}",
                "score": 0.9 - (i * 0.1),
                "content_hash": doc_hash
            })

        return results

    @staticmethod
    def verify_determinism(query: str, filters: Dict[str, Any], index_hash: str) -> str:
        """
        Returns a digest of the top-K results to verify determinism.
        """
        results = RetrieverContract.retrieve(query, filters, index_hash)
        # De-dup by content_hash is implicit if retrieval is deterministic,
        # but we can enforce it here if needed.

        # Serialize results for hashing
        return hashlib.blake2b(json.dumps(results, sort_keys=True).encode()).hexdigest()
```

src/farfan_pipeline/infrastructure/contractual/dura_lex/risk_certificate.py

```
"""
Risk Certificate Contract (RCC) - Implementation
"""
import numpy as np
from typing import List, Tuple, Dict

class RiskCertificateContract:
    @staticmethod
    def conformal_prediction(
        calibration_scores: List[float],
        alpha: float
    ) -> float:
        """
        Computes the quantile for conformal prediction.
        q = (1 - alpha) * (n + 1) / n corrected quantile
        """
        n = len(calibration_scores)
        q_level = np.ceil((n + 1) * (1 - alpha)) / n
        q_level = min(1.0, max(0.0, q_level))

        # Use numpy quantile
        return np.quantile(calibration_scores, q_level, method='higher')

    @staticmethod
    def verify_risk(
        calibration_data: List[float],
        holdout_data: List[float],
        alpha: float,
        seed: int
    ) -> Dict[str, float]:
        """
        Verifies that empirical coverage is approx (1-alpha) and risk <= alpha.
        """
        np.random.seed(seed)

        # Compute threshold from calibration data
             threshold  =  RiskCertificateContract.conformal_prediction(calibration_data,
alpha)

        # Check coverage on holdout
        covered = [s <= threshold for s in holdout_data]
        coverage = sum(covered) / len(holdout_data)
        risk = 1.0 - coverage

        return {
            "alpha": alpha,
            "threshold": float(threshold),
            "coverage": coverage,
            "risk": risk
        }
```