src/farfan_pipeline/infrastructure/__init__.py

```python
"""
Cross-Cutting Infrastructure
=============================

This package contains cross-cutting infrastructure components used across
all phases of the F.A.R.F.A.N pipeline:

- SISAS: Signal Intelligence System for Advanced Structuring
- CAPAZ: Calibration and Parametrization System
- Dura Lex: Contract enforcement and runtime validation
"""

__all__ = [
    "irrigation_using_signals",
    "capaz_calibration_parmetrization",
    "contractual",
]
```

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

```python
# Core contracts __init__.py
"""Core contracts package for runtime contract infrastructure."""

from .runtime_contracts import SegmentationMethod, SegmentationInfo, FallbackCategory

from .audit_trail import (
    VerificationManifest,
    CalibrationScore,
    ParametrizationConfig,
    DeterminismSeeds,
    ResultsBundle,
    OperationTrace,
    generate_manifest,
    verify_manifest,
    reconstruct_score,
    validate_determinism,
    StructuredAuditLogger,
    TraceGenerator,
    create_trace_example,
)

__all__ = [
    "SegmentationMethod",
    "SegmentationInfo",
    "FallbackCategory",
    "VerificationManifest",
    "CalibrationScore",
    "ParametrizationConfig",
    "DeterminismSeeds",
    "ResultsBundle",
    "OperationTrace",
    "generate_manifest",
    "verify_manifest",
    "reconstruct_score",
    "validate_determinism",
    "StructuredAuditLogger",
    "TraceGenerator",
    "create_trace_example",
]
```

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

```python
"""
Aggregation Contract - Dura Lex Enforcement for Phases 4-7

This module defines formal contracts for the hierarchical aggregation pipeline:
- Phase 4: DimensionAggregator (5 micro ? 1 dimension)
- Phase 5: AreaPolicyAggregator (6 dimensions ? 1 area)
- Phase 6: ClusterAggregator (multiple areas ? 1 cluster)
- Phase 7: MacroAggregator (4 clusters ? 1 holistic score)

Contract Hierarchy:
    BaseAggregationContract
        ??? DimensionAggregationContract
        ??? AreaAggregationContract
        ??? ClusterAggregationContract
        ??? MacroAggregationContract

Mathematical Invariants:
    [AGG-001] Weight normalization: ?(weights) = 1.0 ± 1e-6
    [AGG-002] Score bounds: 0.0 ? score ? MAX_SCORE
    [AGG-003] Coherence bounds: 0.0 ? coherence ? 1.0
    [AGG-004] Hermeticity: No gaps, no overlaps, no duplicates
    [AGG-005] Monotonicity: More high scores ? higher aggregate
    [AGG-006] Convexity: weighted_avg lies in convex hull of inputs
"""

from __future__ import annotations

import logging
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Any

logger = logging.getLogger(__name__)


@dataclass(frozen=True)
class AggregationContractViolation:
    """Records contract violation with severity and context."""

    contract_id: str
    invariant_id: str
    severity: str  # CRITICAL, HIGH, MEDIUM, LOW
    message: str
    actual_value: Any
    expected_value: Any
    context: dict[str, Any]


class BaseAggregationContract(ABC):
    """
    Base contract for all aggregation operations.
```

```
Enforces fundamental mathematical properties:
- Weight normalization
- Score bounds
- Non-negativity
- Reproducibility
"""

WEIGHT_TOLERANCE = 1e-6
MAX_SCORE = 3.0
MIN_SCORE = 0.0

def __init__(self, contract_id: str, abort_on_violation: bool = True):
    """
    Initialize aggregation contract.

    Args:
        contract_id: Unique identifier for contract
        abort_on_violation: Whether to raise exception on violation
    """
    self.contract_id = contract_id
    self.abort_on_violation = abort_on_violation
    self.violations: list[AggregationContractViolation] = []

def validate_weight_normalization(
    self,
    weights: list[float],
    context: dict[str, Any] | None = None
) -> bool:
    """
    [AGG-001] Validate weights sum to 1.0 within tolerance.

    Mathematical property: ?(w_i) = 1.0 ± ? where ? = 1e-6
    Rationale: Normalized weights ensure proper probability distribution

    Args:
        weights: List of weights to validate
        context: Additional context for error reporting

    Returns:
        True if valid, False otherwise

    Raises:
        ValueError: If abort_on_violation and validation fails
    """
    if not weights:
        violation = AggregationContractViolation(
            contract_id=self.contract_id,
            invariant_id="AGG-001",
            severity="CRITICAL",
            message="Empty weights list",
            actual_value=[],
            expected_value="non-empty list",
            context=context or {}
        )
```

```
            self.violations.append(violation)
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        weight_sum = sum(weights)
        expected = 1.0

        if abs(weight_sum - expected) > self.WEIGHT_TOLERANCE:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-001",
                severity="CRITICAL",
                message=f"Weights do not sum to 1.0: sum={weight_sum:.10f}",
                actual_value=weight_sum,
                expected_value=expected,
                context=context or {}
            )
            self.violations.append(violation)
                    logger.error(f"[{self.contract_id}] Weight normalization violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True

    def validate_score_bounds(
        self,
        score: float,
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        [AGG-002] Validate score within valid bounds.

        Mathematical property: MIN_SCORE ? score ? MAX_SCORE
        Rationale: Scores must lie in defined range for consistency

        Args:
            score: Score to validate
            context: Additional context for error reporting

        Returns:
            True if valid, False otherwise

        Raises:
            ValueError: If abort_on_violation and validation fails
        """
        if not (self.MIN_SCORE <= score <= self.MAX_SCORE):
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-002",
                severity="HIGH",
                   message=f"Score out of bounds: {score:.4f} not in [{self.MIN_SCORE},
{self.MAX_SCORE}]",
                actual_value=score,
                expected_value=f"[{self.MIN_SCORE}, {self.MAX_SCORE}]",
                context=context or {}
            )
            self.violations.append(violation)
                        logger.warning(f"[{self.contract_id}]  Score  bounds  violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True

    def validate_coherence_bounds(
        self,
        coherence: float,
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        [AGG-003] Validate coherence within [0, 1] bounds.

        Mathematical property: 0.0 ? coherence ? 1.0
        Rationale: Coherence is a normalized metric

        Args:
            coherence: Coherence value to validate
            context: Additional context for error reporting

        Returns:
            True if valid, False otherwise

        Raises:
            ValueError: If abort_on_violation and validation fails
        """
        if not (0.0 <= coherence <= 1.0):
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-003",
                severity="MEDIUM",
                message=f"Coherence out of bounds: {coherence:.4f} not in [0, 1]",
                actual_value=coherence,
                expected_value="[0.0, 1.0]",
                context=context or {}
            )
            self.violations.append(violation)
                    logger.warning(f"[{self.contract_id}]  Coherence  bounds  violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True
    def validate_convexity(
        self,
        aggregated: float,
        inputs: list[float],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        [AGG-006] Validate convexity: aggregated in convex hull of inputs.

        Mathematical property: min(inputs) ? aggregated ? max(inputs)
        Rationale: Weighted averages lie within input bounds

        Args:
            aggregated: Aggregated result
            inputs: Input scores
            context: Additional context for error reporting

        Returns:
            True if valid, False otherwise
        """
        if not inputs:
            return True

        min_input = min(inputs)
        max_input = max(inputs)

        # Allow small tolerance for floating point
        tolerance = 1e-6

        if not (min_input - tolerance <= aggregated <= max_input + tolerance):
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-006",
                severity="HIGH",
                 message=f"Convexity violation: {aggregated:.4f} not in [{min_input:.4f},
{max_input:.4f}]",
                actual_value=aggregated,
                expected_value=f"[{min_input:.4f}, {max_input:.4f}]",
                context=context or {}
            )
            self.violations.append(violation)
                            logger.warning(f"[{self.contract_id}]  Convexity  violation:
{violation.message}")
            # Don't abort on convexity - it's informational
            return False

        return True

    @abstractmethod
    def validate_hermeticity(
        self,
        actual_ids: set[str],
        expected_ids: set[str],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        [AGG-004] Validate hermeticity: no gaps, no overlaps, no duplicates.

        Must be implemented by subclasses with specific validation logic.
        """
        pass

    def get_violations(self) -> list[AggregationContractViolation]:
        """Return all recorded violations."""
        return self.violations.copy()

    def clear_violations(self) -> None:
        """Clear all recorded violations."""
        self.violations.clear()


class DimensionAggregationContract(BaseAggregationContract):
    """
    Contract for Phase 4: Dimension Aggregation.

    Validates:
    - 5 micro questions per dimension (Q1-Q5)
    - Weight normalization
    - Score bounds
    - No duplicate questions
    """

    EXPECTED_QUESTION_COUNT = 5

    def __init__(self, abort_on_violation: bool = True):
        super().__init__("DIM_AGG", abort_on_violation)

    def validate_hermeticity(
        self,
        actual_ids: set[str],
        expected_ids: set[str],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        Validate dimension has exactly the expected questions.

        Args:
            actual_ids: Actual question IDs present
            expected_ids: Expected question IDs
            context: Additional context

        Returns:
            True if hermetic, False otherwise
        """
        missing = expected_ids - actual_ids
        extra = actual_ids - expected_ids

        if missing or extra:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-004",
                severity="HIGH",
                          message=f"Dimension  hermeticity  violation:  missing={missing},
extra={extra}",
                actual_value=actual_ids,
                expected_value=expected_ids,
                context=context or {}
            )
            self.violations.append(violation)
                            logger.error(f"[{self.contract_id}]  Hermeticity  violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True


class AreaAggregationContract(BaseAggregationContract):
    """
    Contract for Phase 5: Policy Area Aggregation.

    Validates:
    - 6 dimensions per area (DIM01-DIM06)
    - Dimension hermeticity
    - No dimension gaps or overlaps
    """

    EXPECTED_DIMENSION_COUNT = 6

    def __init__(self, abort_on_violation: bool = True):
        super().__init__("AREA_AGG", abort_on_violation)

    def validate_hermeticity(
        self,
        actual_ids: set[str],
        expected_ids: set[str],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        Validate area has exactly the expected dimensions.

        Args:
            actual_ids: Actual dimension IDs present
            expected_ids: Expected dimension IDs
            context: Additional context

        Returns:
            True if hermetic, False otherwise
        """
        missing = expected_ids - actual_ids
        extra = actual_ids - expected_ids
        if missing or extra:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-004",
                severity="CRITICAL",
                message=f"Area hermeticity violation: missing={missing}, extra={extra}",
                actual_value=actual_ids,
                expected_value=expected_ids,
                context=context or {}
            )
            self.violations.append(violation)
                            logger.error(f"[{self.contract_id}]  Hermeticity  violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True


class ClusterAggregationContract(BaseAggregationContract):
    """
    Contract for Phase 6: Cluster (MESO) Aggregation.

    Validates:
    - Variable areas per cluster (based on cluster definition)
    - Coherence calculation
    - Penalty application
    """

    def __init__(self, abort_on_violation: bool = True):
        super().__init__("CLUSTER_AGG", abort_on_violation)

    def validate_hermeticity(
        self,
        actual_ids: set[str],
        expected_ids: set[str],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        Validate cluster has exactly the expected policy areas.

        Args:
            actual_ids: Actual area IDs present
            expected_ids: Expected area IDs for this cluster
            context: Additional context

        Returns:
            True if hermetic, False otherwise
        """
        missing = expected_ids - actual_ids
        extra = actual_ids - expected_ids
        if missing or extra:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-004",
                severity="HIGH",
                           message=f"Cluster  hermeticity  violation:  missing={missing},
extra={extra}",
                actual_value=actual_ids,
                expected_value=expected_ids,
                context=context or {}
            )
            self.violations.append(violation)
                            logger.error(f"[{self.contract_id}]  Hermeticity  violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True


class MacroAggregationContract(BaseAggregationContract):
    """
    Contract for Phase 7: Macro (Holistic) Aggregation.

    Validates:
    - 4 clusters (MESO questions)
    - Cross-cutting coherence
    - Strategic alignment
    """

    EXPECTED_CLUSTER_COUNT = 4

    def __init__(self, abort_on_violation: bool = True):
        super().__init__("MACRO_AGG", abort_on_violation)

    def validate_hermeticity(
        self,
        actual_ids: set[str],
        expected_ids: set[str],
        context: dict[str, Any] | None = None
    ) -> bool:
        """
        Validate macro has exactly 4 clusters.

        Args:
            actual_ids: Actual cluster IDs present
            expected_ids: Expected cluster IDs (should be 4)
            context: Additional context

        Returns:
            True if hermetic, False otherwise
        """
        if len(actual_ids) != self.EXPECTED_CLUSTER_COUNT:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-004",
                severity="CRITICAL",
                message=f"Macro hermeticity violation: expected
{self.EXPECTED_CLUSTER_COUNT} clusters, got {len(actual_ids)}",
                actual_value=actual_ids,
                expected_value=expected_ids,
                context=context or {}
            )
            self.violations.append(violation)
            logger.error(f"[{self.contract_id}] Hermeticity violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        missing = expected_ids - actual_ids
        extra = actual_ids - expected_ids

        if missing or extra:
            violation = AggregationContractViolation(
                contract_id=self.contract_id,
                invariant_id="AGG-004",
                severity="CRITICAL",
                message=f"Macro hermeticity violation: missing={missing},
extra={extra}",
                actual_value=actual_ids,
                expected_value=expected_ids,
                context=context or {}
            )
            self.violations.append(violation)
            logger.error(f"[{self.contract_id}] Hermeticity violation:
{violation.message}")
            if self.abort_on_violation:
                raise ValueError(f"[{self.contract_id}] {violation.message}")
            return False

        return True


# Factory function for contract creation
def create_aggregation_contract(
    level: str,
    abort_on_violation: bool = True
) -> BaseAggregationContract:
    """
    Factory function to create appropriate aggregation contract.

    Args:
        level: Aggregation level (dimension, area, cluster, macro)
        abort_on_violation: Whether to abort on contract violations

    Returns:
        Appropriate contract instance

    Raises:
        ValueError: If level is invalid
    """
    contracts = {
        "dimension": DimensionAggregationContract,
        "area": AreaAggregationContract,
        "cluster": ClusterAggregationContract,
        "macro": MacroAggregationContract,
    }

    if level.lower() not in contracts:
            raise ValueError(f"Invalid aggregation level: {level}. Must be one of
{list(contracts.keys())}")

    return contracts[level.lower()](abort_on_violation=abort_on_violation)
```

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

```python
"""
Alignment Stability Contract (ASC) - Implementation
"""
import hashlib
import json
from typing import List, Dict, Any, Tuple

class AlignmentStabilityContract:
    @staticmethod
    def compute_alignment(
        sections: List[str],
        standards: List[str],
        params: Dict[str, Any]
    ) -> Dict[str, Any]:
        """
        Simulates Optimal Transport (EGW) alignment.
        In a real system, this would use POT (Python Optimal Transport).
        """
        # Deterministic simulation based on inputs
        input_str = f"{sections}:{standards}:{json.dumps(params, sort_keys=True)}"
        hasher = hashlib.blake2b(input_str.encode(), digest_size=32)
        digest = hasher.hexdigest()

        # Simulate a plan (matrix) digest
        plan_digest = hashlib.blake2b(f"plan_{digest}".encode()).hexdigest()

        # Simulate cost and unmatched mass
        cost = int(digest[:4], 16) / 1000.0
        unmatched_mass = int(digest[4:8], 16) / 10000.0

        return {
            "plan_digest": plan_digest,
            "cost": cost,
            "unmatched_mass": unmatched_mass
        }

    @staticmethod
    def verify_stability(
        sections: List[str],
        standards: List[str],
        params: Dict[str, Any]
    ) -> bool:
        """
        Verifies reproducibility with fixed hyperparameters.
        """
        res1 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
        res2 = AlignmentStabilityContract.compute_alignment(sections, standards, params)
        return res1 == res2
```

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

```
#!/usr/bin/env python3
"""
Comprehensive audit of 300 executor contracts for completeness, alignment, and signal
wiring.

This script audits all contracts in
src/canonic_phases/Phase_two/json_files_phase_two/executor_contracts/specialized/
to ensure:
1. Completeness and disaggregation of all required fields per v3 schema
2. Alignment with base_executor_with_contract.py requirements
3. Proper wiring between EvidenceAssembler, EvidenceRegistry, and EvidenceValidator
4. Correct signal irrigation and synchronization flow
"""

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

# Base requirements from base_executor_with_contract.py
V3_REQUIRED_TOP_LEVEL_FIELDS = [
    "identity",
    "executor_binding",
    "method_binding",
    "question_context",
    "evidence_assembly",
    "validation_rules",
    "error_handling",
]

V3_IDENTITY_FIELDS = [
    "base_slot",
    "question_id",
    "dimension_id",
    "policy_area_id",
    "contract_version",
]

V3_METHOD_BINDING_FIELDS = [
    "orchestration_mode",
]

V3_EVIDENCE_ASSEMBLY_FIELDS = [
    "assembly_rules",
]

V3_ERROR_HANDLING_FIELDS = [
    "failure_contract",
    "on_method_failure",
]
```

```
# Signal irrigation requirements
SIGNAL_REQUIREMENTS_FIELDS = [
    "mandatory_signals",
    "optional_signals",
    "minimum_signal_threshold",
]

# Assembly rule requirements for EvidenceAssembler
ASSEMBLY_RULE_FIELDS = [
    "target",
    "sources",
    "merge_strategy",
]

# Validation rule requirements for EvidenceValidator
VALIDATION_RULE_FIELDS = [
    "field",
    "required",
]


class ContractAuditor:
    """Auditor for executor contracts completeness and alignment."""

    def __init__(self, contracts_dir: Path):
        self.contracts_dir = contracts_dir
        self.results = {
            "total_contracts": 0,
            "passed": 0,
            "failed": 0,
            "errors": [],
            "warnings": [],
            "stats": defaultdict(int),
        }

    def audit_all_contracts(self) -> Dict[str, Any]:
        """Audit all contracts in the directory."""
        contract_files = sorted(self.contracts_dir.glob("Q*.v3.json"))
        self.results["total_contracts"] = len(contract_files)

        print(f"Auditing {len(contract_files)} contracts...")

        for contract_file in contract_files:
            self._audit_single_contract(contract_file)

        # Calculate summary statistics
        self._calculate_summary_stats()

        return self.results

    def _audit_single_contract(self, contract_file: Path) -> None:
        """Audit a single contract file."""
        contract_id = contract_file.stem
```

```
try:
    with open(contract_file, "r", encoding="utf-8") as f:
        contract = json.load(f)
except json.JSONDecodeError as e:
    self.results["failed"] += 1
    self.results["errors"].append(
        f"[{contract_id}] Invalid JSON: {e}"
    )
    return
except Exception as e:
    self.results["failed"] += 1
    self.results["errors"].append(
        f"[{contract_id}] Failed to read file: {e}"
    )
    return

# Audit checks
errors = []
warnings = []

# 1. Check v3 top-level completeness
errors.extend(self._check_top_level_fields(contract, contract_id))

# 2. Check identity section
errors.extend(self._check_identity_section(contract, contract_id))

# 3. Check method_binding completeness and disaggregation
errors.extend(self._check_method_binding(contract, contract_id))

# 4. Check evidence_assembly alignment with EvidenceAssembler
errors.extend(self._check_evidence_assembly(contract, contract_id))

# 5. Check validation_rules alignment with EvidenceValidator
warnings.extend(self._check_validation_rules(contract, contract_id))

# 6. Check signal_requirements for irrigation
errors.extend(self._check_signal_requirements(contract, contract_id))

# 7. Check error_handling for failure_contract wiring
errors.extend(self._check_error_handling(contract, contract_id))

# 8. Check question_context completeness
errors.extend(self._check_question_context(contract, contract_id))

# Store results
if errors:
    self.results["failed"] += 1
    self.results["errors"].extend(errors)
else:
    self.results["passed"] += 1

if warnings:
    self.results["warnings"].extend(warnings)
```

```
        # Update statistics
        self._update_stats(contract)

    def _check_top_level_fields(self, contract: Dict, contract_id: str) -> List[str]:
        """Check that all v3 top-level fields are present."""
        errors = []
        for field in V3_REQUIRED_TOP_LEVEL_FIELDS:
            if field not in contract:
                errors.append(
                    f"[{contract_id}] Missing required top-level field: {field}"
                )
        return errors

    def _check_identity_section(self, contract: Dict, contract_id: str) -> List[str]:
        """Check identity section completeness."""
        errors = []
        identity = contract.get("identity", {})

        for field in V3_IDENTITY_FIELDS:
            if field not in identity:
                errors.append(
                    f"[{contract_id}] Missing identity field: {field}"
                )

        # Verify base_slot matches expected format
        if "base_slot" in identity:
            base_slot = identity["base_slot"]
            if not base_slot or not base_slot.startswith("D"):
                errors.append(
                    f"[{contract_id}] Invalid base_slot format: {base_slot}"
                )

        return errors

    def _check_method_binding(self, contract: Dict, contract_id: str) -> List[str]:
        """Check method_binding completeness and disaggregation."""
        errors = []
        method_binding = contract.get("method_binding", {})

        for field in V3_METHOD_BINDING_FIELDS:
            if field not in method_binding:
                errors.append(
                    f"[{contract_id}] Missing method_binding field: {field}"
                )

        orchestration_mode = method_binding.get("orchestration_mode")

        if orchestration_mode == "multi_method_pipeline":
            # Check methods array exists and is properly disaggregated
            if "methods" not in method_binding:
                errors.append(
                     f"[{contract_id}] Missing 'methods' array for multi_method_pipeline
mode"
                )
            elif not isinstance(method_binding["methods"], list):
                errors.append(
                    f"[{contract_id}] 'methods' must be a list"
                )
            else:
                methods = method_binding["methods"]
                if not methods:
                    errors.append(
                        f"[{contract_id}] Empty methods array"
                    )

                # Check disaggregation of each method
                for idx, method_spec in enumerate(methods):
                    if not isinstance(method_spec, dict):
                        errors.append(
                            f"[{contract_id}] methods[{idx}] is not a dict"
                        )
                        continue

                    # Required method fields
                    for req_field in ["class_name", "method_name", "provides",
"priority"]:
                        if req_field not in method_spec:
                            errors.append(
                                f"[{contract_id}] methods[{idx}] missing '{req_field}'"
                            )
        else:
            # Single method mode - check class_name and method_name
            if "class_name" not in method_binding and "primary_method" not in
method_binding:
                errors.append(
                    f"[{contract_id}] Missing class_name or primary_method for
single_method mode"
                )

        return errors

    def _check_evidence_assembly(self, contract: Dict, contract_id: str) -> List[str]:
        """Check evidence_assembly alignment with EvidenceAssembler."""
        errors = []
        evidence_assembly = contract.get("evidence_assembly", {})

        for field in V3_EVIDENCE_ASSEMBLY_FIELDS:
            if field not in evidence_assembly:
                errors.append(
                    f"[{contract_id}] Missing evidence_assembly field: {field}"
                )

        assembly_rules = evidence_assembly.get("assembly_rules", [])
        if not isinstance(assembly_rules, list):
            errors.append(
                f"[{contract_id}] assembly_rules must be a list"
            )
        elif not assembly_rules:
            errors.append(
                f"[{contract_id}] Empty assembly_rules - no evidence will be assembled"
            )
        else:
            # Check each assembly rule structure
            for idx, rule in enumerate(assembly_rules):
                if not isinstance(rule, dict):
                    errors.append(
                        f"[{contract_id}] assembly_rules[{idx}] is not a dict"
                    )
                    continue

                for req_field in ASSEMBLY_RULE_FIELDS:
                    if req_field not in rule:
                        errors.append(
                                         f"[{contract_id}]  assembly_rules[{idx}]  missing
'{req_field}'"
                        )

                # Validate merge_strategy is supported by EvidenceAssembler
                merge_strategy = rule.get("merge_strategy", "")
                valid_strategies = {
                    "concat", "first", "last", "mean", "max", "min",
                    "weighted_mean", "majority"
                }
                if merge_strategy and merge_strategy not in valid_strategies:
                    errors.append(
                          f"[{contract_id}] assembly_rules[{idx}] invalid merge_strategy
'{merge_strategy}'"
                    )

        return errors

    def _check_validation_rules(self, contract: Dict, contract_id: str) -> List[str]:
        """Check validation_rules alignment with EvidenceValidator."""
        warnings = []
        validation_rules_section = contract.get("validation_rules", {})

        if not isinstance(validation_rules_section, dict):
            return [f"[{contract_id}] validation_rules must be a dict"]

        rules = validation_rules_section.get("rules", [])
        if not isinstance(rules, list):
            warnings.append(
                f"[{contract_id}] validation_rules.rules must be a list"
            )
        elif not rules:
            warnings.append(
                f"[{contract_id}] Empty validation rules - no validation will occur"
            )
        else:
            # Check each validation rule structure
            for idx, rule in enumerate(rules):
                if not isinstance(rule, dict):
```

```
                warnings.append(
                    f"[{contract_id}] validation_rules.rules[{idx}] is not a dict"
                )
                continue

            if "field" not in rule:
                warnings.append(
                    f"[{contract_id}] validation_rules.rules[{idx}] missing 'field'"
                )

        # Check na_policy
        na_policy = validation_rules_section.get("na_policy")
        valid_na_policies = {"abort_on_critical", "score_zero", "propagate"}
        if na_policy and na_policy not in valid_na_policies:
            warnings.append(
                f"[{contract_id}] Invalid na_policy '{na_policy}'"
            )

        return warnings

    def _check_signal_requirements(self, contract: Dict, contract_id: str) -> List[str]:
        """Check signal_requirements for proper irrigation."""
        errors = []
        signal_requirements = contract.get("signal_requirements", {})

        if not signal_requirements:
            errors.append(
                f"[{contract_id}] Missing signal_requirements section"
            )
            return errors

        if not isinstance(signal_requirements, dict):
            errors.append(
                f"[{contract_id}] signal_requirements must be a dict"
            )
            return errors

        # Check mandatory_signals structure
        if "mandatory_signals" in signal_requirements:
            mandatory = signal_requirements["mandatory_signals"]
            if not isinstance(mandatory, list):
                errors.append(
                    f"[{contract_id}] mandatory_signals must be a list"
                )

        # Check optional_signals structure
        if "optional_signals" in signal_requirements:
            optional = signal_requirements["optional_signals"]
            if not isinstance(optional, list):
                errors.append(
                    f"[{contract_id}] optional_signals must be a list"
                )

        # Check minimum_signal_threshold
        if "minimum_signal_threshold" in signal_requirements:
            threshold = signal_requirements["minimum_signal_threshold"]
             if not isinstance(threshold, (int, float)) or threshold < 0 or threshold >
1:
                errors.append(
                    f"[{contract_id}] minimum_signal_threshold must be between 0 and 1"
                )

        return errors

    def _check_error_handling(self, contract: Dict, contract_id: str) -> List[str]:
        """Check error_handling for failure_contract wiring."""
        errors = []
        error_handling = contract.get("error_handling", {})

        if not error_handling:
            errors.append(
                f"[{contract_id}] Missing error_handling section"
            )
            return errors

        # Check failure_contract exists for validator wiring
        if "failure_contract" not in error_handling:
            errors.append(
                f"[{contract_id}] Missing failure_contract in error_handling"
            )
        else:
            failure_contract = error_handling["failure_contract"]
            if not isinstance(failure_contract, dict):
                errors.append(
                    f"[{contract_id}] failure_contract must be a dict"
                )
            else:
                # Check abort_if conditions for validator
                if "abort_if" in failure_contract:
                    abort_if = failure_contract["abort_if"]
                    if not isinstance(abort_if, list):
                        errors.append(
                            f"[{contract_id}] failure_contract.abort_if must be a list"
                        )

                # Check emit_code for signal propagation
                if "emit_code" not in failure_contract:
                    errors.append(
                        f"[{contract_id}] failure_contract missing 'emit_code'"
                    )

        # Check on_method_failure
        if "on_method_failure" not in error_handling:
            errors.append(
                f"[{contract_id}] Missing on_method_failure in error_handling"
            )

        return errors
```

```
def _check_question_context(self, contract: Dict, contract_id: str) -> List[str]:
    """Check question_context completeness."""
    errors = []
    question_context = contract.get("question_context", {})

    if not question_context:
        errors.append(
            f"[{contract_id}] Missing question_context section"
        )
        return errors

    # Check expected_elements for evidence extraction
    if "expected_elements" not in question_context:
        errors.append(
            f"[{contract_id}] Missing expected_elements in question_context"
        )
    elif not isinstance(question_context["expected_elements"], list):
        errors.append(
            f"[{contract_id}] expected_elements must be a list"
        )

    # Check patterns for signal irrigation
    if "patterns" not in question_context:
        errors.append(
            f"[{contract_id}] Missing patterns in question_context"
        )
    elif not isinstance(question_context["patterns"], list):
        errors.append(
            f"[{contract_id}] patterns must be a list"
        )

    return errors

def _update_stats(self, contract: Dict) -> None:
    """Update statistics from contract."""
    # Count orchestration modes
    method_binding = contract.get("method_binding", {})
    orchestration_mode = method_binding.get("orchestration_mode", "unknown")
    self.results["stats"][f"orchestration_mode_{orchestration_mode}"] += 1

    # Count methods in multi-method pipelines
    if orchestration_mode == "multi_method_pipeline":
        methods_count = len(method_binding.get("methods", []))
        self.results["stats"]["total_methods"] += methods_count

    # Count assembly rules
    evidence_assembly = contract.get("evidence_assembly", {})
    assembly_rules_count = len(evidence_assembly.get("assembly_rules", []))
    self.results["stats"]["total_assembly_rules"] += assembly_rules_count

    # Count validation rules
    validation_rules_section = contract.get("validation_rules", {})
    validation_rules_count = len(validation_rules_section.get("rules", []))
```

```
        self.results["stats"]["total_validation_rules"] += validation_rules_count

        # Count signal requirements
        signal_requirements = contract.get("signal_requirements", {})
        if signal_requirements:
            self.results["stats"]["contracts_with_signal_requirements"] += 1
            mandatory_signals = len(signal_requirements.get("mandatory_signals", []))
            optional_signals = len(signal_requirements.get("optional_signals", []))
            self.results["stats"]["total_mandatory_signals"] += mandatory_signals
            self.results["stats"]["total_optional_signals"] += optional_signals

    def _calculate_summary_stats(self) -> None:
        """Calculate summary statistics."""
        stats = self.results["stats"]

        # Calculate averages
        if stats.get("orchestration_mode_multi_method_pipeline", 0) > 0:
            stats["avg_methods_per_pipeline"] = (
                stats["total_methods"] /
                stats["orchestration_mode_multi_method_pipeline"]
            )

        if self.results["total_contracts"] > 0:
            stats["avg_assembly_rules_per_contract"] = (
                stats["total_assembly_rules"] /
                self.results["total_contracts"]
            )
            stats["avg_validation_rules_per_contract"] = (
                stats["total_validation_rules"] /
                self.results["total_contracts"]
            )
            stats["signal_requirements_coverage"] = (
                stats["contracts_with_signal_requirements"] /
                self.results["total_contracts"] * 100
            )

    def print_report(self) -> None:
        """Print formatted audit report."""
        print("\n" + "="*80)
        print("AUDIT REPORT: 300 Executor Contracts")
        print("="*80)

        print(f"\nTotal Contracts Audited: {self.results['total_contracts']}")
        print(f"? Passed: {self.results['passed']}")
        print(f"? Failed: {self.results['failed']}")

        if self.results['passed'] == self.results['total_contracts']:
            print("\n? ALL CONTRACTS PASSED AUDIT!")
        else:
            pass_rate = (self.results['passed'] / self.results['total_contracts']) * 100
            print(f"\n? Pass Rate: {pass_rate:.1f}%")

        # Print statistics
        print("\n" + "-"*80)
        print("STATISTICS")
        print("-"*80)
        stats = self.results["stats"]

        print(f"Orchestration Modes:")
        print(f"  - Multi-method pipeline: {stats.get('orchestration_mode_multi_method_pipeline', 0)}")
        print(f"  - Single method: {stats.get('orchestration_mode_single_method', 0)}")
        print(f"  - Unknown: {stats.get('orchestration_mode_unknown', 0)}")

        print(f"\nMethod Disaggregation:")
        print(f"  - Total methods: {stats.get('total_methods', 0)}")
        print(f"  - Avg methods/pipeline: {stats.get('avg_methods_per_pipeline', 0):.1f}")

        print(f"\nEvidence Assembly:")
        print(f"  - Total assembly rules: {stats.get('total_assembly_rules', 0)}")
        print(f"  - Avg rules/contract: {stats.get('avg_assembly_rules_per_contract', 0):.1f}")

        print(f"\nValidation:")
        print(f"  - Total validation rules: {stats.get('total_validation_rules', 0)}")
        print(f"  - Avg rules/contract: {stats.get('avg_validation_rules_per_contract', 0):.1f}")

        print(f"\nSignal Irrigation:")
        print(f"  - Contracts with signal_requirements: {stats.get('contracts_with_signal_requirements', 0)}")
        print(f"  - Coverage: {stats.get('signal_requirements_coverage', 0):.1f}%")
        print(f"  - Total mandatory signals: {stats.get('total_mandatory_signals', 0)}")
        print(f"  - Total optional signals: {stats.get('total_optional_signals', 0)}")

        # Print errors
        if self.results["errors"]:
            print("\n" + "-"*80)
            print(f"ERRORS ({len(self.results['errors'])})")
            print("-"*80)
            for error in self.results["errors"][:50]:  # Show first 50
                print(f"  {error}")
            if len(self.results["errors"]) > 50:
                print(f"  ... and {len(self.results['errors']) - 50} more errors")

        # Print warnings
        if self.results["warnings"]:
            print("\n" + "-"*80)
            print(f"WARNINGS ({len(self.results['warnings'])})")
            print("-"*80)
            for warning in self.results["warnings"][:30]:  # Show first 30
                print(f"  {warning}")
            if len(self.results["warnings"]) > 30:
                print(f"  ... and {len(self.results['warnings']) - 30} more warnings")

        print("\n" + "="*80)
```

```
def main():
    """Main entry point."""
    # Find contracts directory
    script_dir = Path(__file__).parent
    contracts_dir = (
        script_dir /
        "src" /
        "canonic_phases" /
        "Phase_two" /
        "json_files_phase_two" /
        "executor_contracts" /
        "specialized"
    )

    if not contracts_dir.exists():
        print(f"Error: Contracts directory not found: {contracts_dir}")
        sys.exit(1)

    # Run audit
    auditor = ContractAuditor(contracts_dir)
    results = auditor.audit_all_contracts()
    auditor.print_report()

    # Save detailed report to JSON
    report_file = script_dir / "audit_contracts_report.json"
    with open(report_file, "w", encoding="utf-8") as f:
        json.dump(results, f, indent=2, ensure_ascii=False)
    print(f"\n? Detailed report saved to: {report_file}")

    # Exit with error code if any contracts failed
    sys.exit(0 if results["failed"] == 0 else 1)


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

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

```python
"""
Audit Trail System with Verification Manifest

Implements comprehensive audit trail for calibration with:
- VerificationManifest dataclass capturing all calibration inputs/outputs
- HMAC signature verification for tamper detection
- Score reconstruction for determinism validation
- Structured JSON logging with rotation and compression
- Operation trace generation for mathematical operations
"""

from __future__ import annotations

import hashlib
import hmac
import json
import logging
import traceback
from dataclasses import dataclass, field, asdict
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

import numpy as np


@dataclass(frozen=True)
class CalibrationScore:
    """Individual method calibration score Cal(I)"""
    method_id: str
    score: float
    confidence: float
    timestamp: str
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass(frozen=True)
class ParametrizationConfig:
    """Parametrization configuration snapshot"""
    config_hash: str
    retry: int
    timeout_s: float
    temperature: float
    thresholds: dict[str, float]
    additional_params: dict[str, Any] = field(default_factory=dict)


@dataclass(frozen=True)
class DeterminismSeeds:
    """Determinism seed tracking"""
    random_seed: int
    numpy_seed: int
    seed_version: str
    base_seed: int | None = None
    policy_unit_id: str | None = None
    correlation_id: str | None = None


@dataclass(frozen=True)
class ResultsBundle:
    """Analysis results bundle"""
    micro_scores: list[float]
    dimension_scores: dict[str, float]
    area_scores: dict[str, float]
    macro_score: float
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass(frozen=True)
class OperationTrace:
    """Trace of a mathematical operation"""
    operation: str
    inputs: dict[str, Any]
    output: Any
    stack_trace: list[str]
    timestamp: str


@dataclass
class VerificationManifest:
    """
    Verification manifest capturing complete calibration context.

    Captures all inputs, parameters, results, and cryptographic signatures
    for tamper detection and determinism validation.
    """
    calibration_scores: dict[str, CalibrationScore]
    parametrization: ParametrizationConfig
    determinism_seeds: DeterminismSeeds
    results: ResultsBundle
    timestamp: str
    validator_version: str
    signature: str = ""
    trace: list[OperationTrace] = field(default_factory=list)

    def to_dict(self) -> dict[str, Any]:
        """Convert manifest to dictionary"""
        data = asdict(self)
        return data

    def to_json(self, indent: int = 2) -> str:
        """Convert manifest to JSON string"""
        return json.dumps(self.to_dict(), indent=indent, default=str)

    @classmethod
    def from_dict(cls, data: dict[str, Any]) -> VerificationManifest:
        """Create manifest from dictionary"""
        calibration_scores = {
            k: CalibrationScore(**v) for k, v in data["calibration_scores"].items()
        }
        parametrization = ParametrizationConfig(**data["parametrization"])
        determinism_seeds = DeterminismSeeds(**data["determinism_seeds"])
        results = ResultsBundle(**data["results"])

        trace = [
            OperationTrace(**t) for t in data.get("trace", [])
        ]

        return cls(
            calibration_scores=calibration_scores,
            parametrization=parametrization,
            determinism_seeds=determinism_seeds,
            results=results,
            timestamp=data["timestamp"],
            validator_version=data["validator_version"],
            signature=data.get("signature", ""),
            trace=trace
        )


def generate_manifest(
    calibration_scores: dict[str, float],
    config_hash: str,
    retry: int,
    timeout_s: float,
    temperature: float,
    thresholds: dict[str, float],
    random_seed: int,
    numpy_seed: int,
    seed_version: str,
    micro_scores: list[float],
    dimension_scores: dict[str, float],
    area_scores: dict[str, float],
    macro_score: float,
    validator_version: str,
    secret_key: str,
    base_seed: int | None = None,
    policy_unit_id: str | None = None,
    correlation_id: str | None = None,
    additional_params: dict[str, Any] | None = None,
    results_metadata: dict[str, Any] | None = None,
    trace: list[OperationTrace] | None = None
) -> VerificationManifest:
    """
    Generate verification manifest with signature.

    Args:
        calibration_scores: Method ID to Cal(I) score mapping
        config_hash: SHA256 hash of configuration
        retry: Retry count
        timeout_s: Timeout in seconds
        temperature: Temperature parameter
        thresholds: Threshold configuration
        random_seed: Random seed used
        numpy_seed: NumPy seed used
        seed_version: Seed derivation algorithm version
        micro_scores: Micro-level scores
        dimension_scores: Dimension-level scores
        area_scores: Area-level scores
        macro_score: Macro-level score
        validator_version: Validator version string
        secret_key: HMAC secret key
        base_seed: Optional base seed
        policy_unit_id: Optional policy unit identifier
        correlation_id: Optional correlation ID
        additional_params: Optional additional parameters
        results_metadata: Optional results metadata
        trace: Optional operation trace

    Returns:
        VerificationManifest with HMAC signature
    """
    timestamp = datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')

    calibration_score_objs = {
        method_id: CalibrationScore(
            method_id=method_id,
            score=score,
            confidence=1.0,
            timestamp=timestamp
        )
        for method_id, score in calibration_scores.items()
    }

    parametrization = ParametrizationConfig(
        config_hash=config_hash,
        retry=retry,
        timeout_s=timeout_s,
        temperature=temperature,
        thresholds=thresholds,
        additional_params=additional_params or {}
    )

    determinism_seeds = DeterminismSeeds(
        random_seed=random_seed,
        numpy_seed=numpy_seed,
        seed_version=seed_version,
        base_seed=base_seed,
        policy_unit_id=policy_unit_id,
        correlation_id=correlation_id
    )

    results = ResultsBundle(
        micro_scores=micro_scores,
        dimension_scores=dimension_scores,
        area_scores=area_scores,
        macro_score=macro_score,
        metadata=results_metadata or {}
    )

    manifest = VerificationManifest(
        calibration_scores=calibration_score_objs,
        parametrization=parametrization,
        determinism_seeds=determinism_seeds,
        results=results,
        timestamp=timestamp,
        validator_version=validator_version,
        trace=trace or []
    )

    signature = _compute_signature(manifest, secret_key)

    object.__setattr__(manifest, 'signature', signature)

    return manifest


def verify_manifest(manifest: VerificationManifest, secret_key: str) -> bool:
    """
    Verify manifest HMAC signature.

    Args:
        manifest: VerificationManifest to verify
        secret_key: HMAC secret key

    Returns:
        True if signature is valid, False otherwise
    """
    if not manifest.signature:
        return False

    provided_signature = manifest.signature

    temp_manifest = VerificationManifest(
        calibration_scores=manifest.calibration_scores,
        parametrization=manifest.parametrization,
        determinism_seeds=manifest.determinism_seeds,
        results=manifest.results,
        timestamp=manifest.timestamp,
        validator_version=manifest.validator_version,
        signature="",
        trace=manifest.trace
    )

    expected_signature = _compute_signature(temp_manifest, secret_key)

    return hmac.compare_digest(provided_signature, expected_signature)
```

```
def reconstruct_score(manifest: VerificationManifest) -> float:
    """
    Reconstruct macro score from manifest to verify computation.

    Args:
        manifest: VerificationManifest with results

    Returns:
        Reconstructed macro score
    """
    if not manifest.results.dimension_scores:
        return 0.0

    dimension_values = list(manifest.results.dimension_scores.values())

    reconstructed = float(np.mean(dimension_values))

    return reconstructed


def validate_determinism(
    manifest1: VerificationManifest,
    manifest2: VerificationManifest
) -> bool:
    """
    Validate determinism: identical inputs should produce identical outputs.

    Args:
        manifest1: First verification manifest
        manifest2: Second verification manifest

    Returns:
        True if deterministic (same seeds ? same results), False otherwise
    """
                              if        (manifest1.determinism_seeds.random_seed        !=
manifest2.determinism_seeds.random_seed or
                                           manifest1.determinism_seeds.numpy_seed      !=
manifest2.determinism_seeds.numpy_seed):
        return True

    if manifest1.parametrization.config_hash != manifest2.parametrization.config_hash:
        return True

    results_match = (
        abs(manifest1.results.macro_score - manifest2.results.macro_score) < 1e-6 and
        manifest1.results.micro_scores == manifest2.results.micro_scores and
        manifest1.results.dimension_scores == manifest2.results.dimension_scores and
        manifest1.results.area_scores == manifest2.results.area_scores
    )

    return results_match
```

```
def _compute_signature(manifest: VerificationManifest, secret_key: str) -> str:
    """
    Compute HMAC-SHA256 signature of manifest.

    Args:
        manifest: VerificationManifest to sign
        secret_key: HMAC secret key

    Returns:
        Hex-encoded HMAC signature
    """
    manifest_dict = manifest.to_dict()

    if "signature" in manifest_dict:
        del manifest_dict["signature"]

    canonical_json = json.dumps(manifest_dict, sort_keys=True, separators=(',', ':'),
default=str)

    signature = hmac.new(
        secret_key.encode('utf-8'),
        canonical_json.encode('utf-8'),
        hashlib.sha256
    )

    return signature.hexdigest()


class StructuredAuditLogger:
    """
    Structured JSON logger for audit trail.

    Logs to logs/calibration/ with daily rotation and compression.
    """

    def __init__(self, log_dir: Path | str = "logs/calibration", component: str =
"audit"):
        """
        Initialize structured audit logger.

        Args:
            log_dir: Directory for log files
            component: Component name for log entries
        """
        self.log_dir = Path(log_dir)
        self.component = component
        self.log_dir.mkdir(parents=True, exist_ok=True)

        self.logger = logging.getLogger(f"audit_trail.{component}")
        self.logger.setLevel(logging.INFO)

        if not self.logger.handlers:
            handler = logging.FileHandler(
                self.log_dir / f"{component}_{self._get_date_suffix()}.log"
            )
            handler.setFormatter(logging.Formatter('%(message)s'))
            self.logger.addHandler(handler)

    def _get_date_suffix(self) -> str:
        """Get date suffix for log rotation"""
        return datetime.now(timezone.utc).strftime("%Y%m%d")

    def log(
        self,
        level: str,
        message: str,
        metadata: dict[str, Any] | None = None
    ) -> None:
        """
        Log structured JSON entry.

        Args:
            level: Log level (INFO, WARNING, ERROR)
            message: Log message
            metadata: Optional metadata dictionary
        """
        entry = {
            "timestamp": datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z'),
            "level": level,
            "component": self.component,
            "message": message,
            "metadata": metadata or {}
        }

        self.logger.info(json.dumps(entry, sort_keys=True, default=str))

    def log_manifest_generation(
        self,
        manifest: VerificationManifest,
        success: bool,
        error: str | None = None
    ) -> None:
        """Log manifest generation event"""
        metadata = {
            "timestamp": manifest.timestamp,
            "validator_version": manifest.validator_version,
            "calibration_count": len(manifest.calibration_scores),
            "macro_score": manifest.results.macro_score,
            "success": success
        }

        if error:
            metadata["error"] = error

        self.log(
            "INFO" if success else "ERROR",
            "Manifest generation completed",
            metadata
        )

    def log_verification(
        self,
        manifest: VerificationManifest,
        verified: bool,
        error: str | None = None
    ) -> None:
        """Log manifest verification event"""
        metadata = {
            "timestamp": manifest.timestamp,
            "validator_version": manifest.validator_version,
            "verified": verified
        }

        if error:
            metadata["error"] = error

        self.log(
            "INFO" if verified else "WARNING",
            "Manifest verification completed",
            metadata
        )

    def log_determinism_check(
        self,
        manifest1: VerificationManifest,
        manifest2: VerificationManifest,
        deterministic: bool
    ) -> None:
        """Log determinism validation event"""
        metadata = {
            "timestamp1": manifest1.timestamp,
            "timestamp2": manifest2.timestamp,
            "deterministic": deterministic,
            "seed_match": (
                manifest1.determinism_seeds.random_seed ==
                manifest2.determinism_seeds.random_seed
            ),
            "config_match": (
                manifest1.parametrization.config_hash ==
                manifest2.parametrization.config_hash
            )
        }

        self.log(
            "INFO" if deterministic else "WARNING",
            "Determinism validation completed",
            metadata
        )


class TraceGenerator:
    """
    Trace generator for mathematical operations.

    Intercepts operations and records inputs, outputs, and stack traces.
    """

    def __init__(self, enabled: bool = True):
        """
        Initialize trace generator.

        Args:
            enabled: Whether tracing is enabled
        """
        self.enabled = enabled
        self.traces: list[OperationTrace] = []

    def trace_operation(
        self,
        operation: str,
        inputs: dict[str, Any],
        output: Any
    ) -> None:
        """
        Trace a mathematical operation.

        Args:
            operation: Operation name (e.g., "np.mean", "aggregate_scores")
            inputs: Input values dictionary
            output: Output value
        """
        if not self.enabled:
            return

        stack = traceback.extract_stack()[:-1]
        stack_trace = [
            f"{frame.filename}:{frame.lineno} in {frame.name}"
            for frame in stack[-5:]
        ]

        trace = OperationTrace(
            operation=operation,
            inputs=inputs,
            output=output,
            stack_trace=stack_trace,
            timestamp=datetime.now(timezone.utc).isoformat().replace('+00:00', 'Z')
        )

        self.traces.append(trace)

    def get_traces(self) -> list[OperationTrace]:
        """Get all recorded traces"""
        return self.traces.copy()

    def clear(self) -> None:
        """Clear all traces"""
        self.traces.clear()

    def __enter__(self) -> "TraceGenerator":
        """Context manager entry"""
        self.clear()
        return self

    def __exit__(self, exc_type: Any, exc_val: Any, exc_tb: Any) -> None:
        """Context manager exit"""
        pass


def create_trace_example(output_dir: Path | str = "trace_examples") -> None:
    """
    Create example trace files for documentation.

    Args:
        output_dir: Directory for example files
    """
    output_path = Path(output_dir)
    output_path.mkdir(parents=True, exist_ok=True)

    trace_gen = TraceGenerator(enabled=True)

    trace_gen.trace_operation(
        "np.mean",
        {"values": [0.8, 0.9, 0.7]},
        0.8
    )

    trace_gen.trace_operation(
        "aggregate_dimension_scores",
        {"dimension_scores": {"DIM01": 0.8, "DIM02": 0.9}},
        0.85
    )

    example_data = {
        "description": "Example operation traces for calibration audit",
        "traces": [asdict(t) for t in trace_gen.get_traces()]
    }

    with open(output_path / "example_traces.json", "w") as f:
        json.dump(example_data, f, indent=2, default=str)
```

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

```
"""
Audit Trail System - Examples and Usage

Demonstrates usage of the audit trail system for calibration verification.
"""


from audit_trail import (
    generate_manifest,
    verify_manifest,
    reconstruct_score,
    validate_determinism,
    StructuredAuditLogger,
    TraceGenerator,
    create_trace_example,
)


def example_basic_manifest_generation():
    """Example: Generate a basic verification manifest"""
    print("=" * 70)
    print("Example 1: Basic Manifest Generation")
    print("=" * 70)

    manifest = generate_manifest(
        calibration_scores={
            "FIN:BayesianNumericalAnalyzer.analyze_numeric_pattern@Q": 0.8004,
            "POL:BayesianEvidenceScorer.compute_bayesian_score@Q": 0.8567,
            "DER:BayesianMechanismInference.infer_mechanism@C": 0.7821,
        },
        config_hash="e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855",
        retry=3,
        timeout_s=300.0,
        temperature=0.7,
        thresholds={
            "min_confidence": 0.7,
            "min_evidence": 0.6,
            "min_coherence": 0.65,
        },
        random_seed=123456,
        numpy_seed=789012,
        seed_version="sha256_v1",
        micro_scores=[0.75, 0.78, 0.82, 0.85, 0.88],
        dimension_scores={
            "DIM01": 0.82,
            "DIM02": 0.89,
            "DIM03": 0.76,
            "DIM04": 0.84,
            "DIM05": 0.78,
            "DIM06": 0.81,
        },
        area_scores={
            "PA01": 0.83,
            "PA02": 0.87,
            "PA03": 0.79,
            "PA04": 0.85,
        },
        macro_score=0.8166666666666667,
        validator_version="2.0.0",
        secret_key="test_secret_key_do_not_use_in_production",
    )

    print("? Manifest generated")
    print(f"  Timestamp: {manifest.timestamp}")
    print(f"  Validator version: {manifest.validator_version}")
    print(f"  Calibration count: {len(manifest.calibration_scores)}")
    print(f"  Macro score: {manifest.results.macro_score}")
    print(f"  Signature: {manifest.signature[:32]}...")
    print()

    return manifest


def example_manifest_verification(manifest):
    """Example: Verify manifest signature"""
    print("=" * 70)
    print("Example 2: Manifest Verification")
    print("=" * 70)

    valid = verify_manifest(manifest, "test_secret_key_do_not_use_in_production")

    print(f"? Signature verification: {'VALID' if valid else 'INVALID'}")

    invalid = verify_manifest(manifest, "wrong_key")
    print(f"? Wrong key verification: {'VALID' if invalid else 'INVALID (expected)'}")
    print()

    return valid


def example_score_reconstruction(manifest):
    """Example: Reconstruct score from manifest"""
    print("=" * 70)
    print("Example 3: Score Reconstruction")
    print("=" * 70)

    reconstructed = reconstruct_score(manifest)
    original = manifest.results.macro_score

    print(f"  Original macro score: {original}")
    print(f"  Reconstructed score:  {reconstructed}")
    print(f"  Difference:           {abs(original - reconstructed)}")
      print(f"? Reconstruction {'PASSED' if abs(original - reconstructed) < 1e-6 else
'FAILED'}")
    print()
```

```
def example_determinism_validation():
    """Example: Validate determinism across runs"""
    print("=" * 70)
    print("Example 4: Determinism Validation")
    print("=" * 70)

    manifest1 = generate_manifest(
        calibration_scores={"method1": 0.8, "method2": 0.9},
        config_hash="abc123",
        retry=3,
        timeout_s=300.0,
        temperature=0.7,
        thresholds={"threshold1": 0.7},
        random_seed=42,
        numpy_seed=42,
        seed_version="sha256_v1",
        micro_scores=[0.8, 0.9],
        dimension_scores={"DIM01": 0.85},
        area_scores={"PA01": 0.85},
        macro_score=0.85,
        validator_version="2.0.0",
        secret_key="test_key",
    )

    manifest2 = generate_manifest(
        calibration_scores={"method1": 0.8, "method2": 0.9},
        config_hash="abc123",
        retry=3,
        timeout_s=300.0,
        temperature=0.7,
        thresholds={"threshold1": 0.7},
        random_seed=42,
        numpy_seed=42,
        seed_version="sha256_v1",
        micro_scores=[0.8, 0.9],
        dimension_scores={"DIM01": 0.85},
        area_scores={"PA01": 0.85},
        macro_score=0.85,
        validator_version="2.0.0",
        secret_key="test_key",
    )

    deterministic = validate_determinism(manifest1, manifest2)

     print(f"? Same seeds, same config, same results: {'DETERMINISTIC' if deterministic
else 'NON-DETERMINISTIC'}")

    manifest3 = generate_manifest(
        calibration_scores={"method1": 0.8, "method2": 0.9},
        config_hash="abc123",
        retry=3,
        timeout_s=300.0,
        temperature=0.7,
        thresholds={"threshold1": 0.7},
        random_seed=99,
        numpy_seed=99,
        seed_version="sha256_v1",
        micro_scores=[0.75, 0.85],
        dimension_scores={"DIM01": 0.80},
        area_scores={"PA01": 0.80},
        macro_score=0.80,
        validator_version="2.0.0",
        secret_key="test_key",
    )

    non_deterministic = validate_determinism(manifest1, manifest3)
    print(f"? Different seeds, different results: {'ALLOWS VARIATION' if not
non_deterministic else 'UNEXPECTED'}")
    print()


def example_structured_logging():
    """Example: Structured logging"""
    print("=" * 70)
    print("Example 5: Structured Logging")
    print("=" * 70)

    logger = StructuredAuditLogger(log_dir="logs/calibration", component="test_audit")

    logger.log("INFO", "Starting calibration", {"phase": "initialization"})
    logger.log("INFO", "Calibration complete", {"phase": "completion", "score": 0.85})
    logger.log("WARNING", "Low confidence detected", {"confidence": 0.62, "threshold":
0.7})

    print("? Logs written to: logs/calibration/test_audit_*.log")
    print("  Log format: Structured JSON with timestamp, level, component, message,
metadata")
    print()


def example_trace_generation():
    """Example: Operation trace generation"""
    print("=" * 70)
    print("Example 6: Operation Trace Generation")
    print("=" * 70)

    with TraceGenerator(enabled=True) as tracer:
        tracer.trace_operation(
            "compute_score",
            {"evidence": 0.85, "confidence": 0.9},
            0.765
        )

        tracer.trace_operation(
            "aggregate_dimensions",
            {"dim_scores": [0.8, 0.9, 0.7]},
            0.8
        )

        traces = tracer.get_traces()

    print(f"? Traced {len(traces)} operations")
    for i, trace in enumerate(traces, 1):
        print(f"  {i}. {trace.operation}: {trace.inputs} ? {trace.output}")
    print()


def example_complete_workflow():
    """Example: Complete audit trail workflow"""
    print("=" * 70)
    print("Example 7: Complete Audit Trail Workflow")
    print("=" * 70)

    logger = StructuredAuditLogger(log_dir="logs/calibration", component="workflow")

    with TraceGenerator(enabled=True) as tracer:
        tracer.trace_operation("initialize", {"seed": 42}, None)
        tracer.trace_operation("compute", {"input": 0.85}, 0.8)

        traces = tracer.get_traces()

    manifest = generate_manifest(
        calibration_scores={"method1": 0.8},
        config_hash="abc",
        retry=3,
        timeout_s=300.0,
        temperature=0.7,
        thresholds={"t1": 0.7},
        random_seed=42,
        numpy_seed=42,
        seed_version="sha256_v1",
        micro_scores=[0.8],
        dimension_scores={"DIM01": 0.8},
        area_scores={"PA01": 0.8},
        macro_score=0.8,
        validator_version="2.0.0",
        secret_key="test_key",
        trace=traces,
    )

    logger.log_manifest_generation(manifest, success=True)

    verified = verify_manifest(manifest, "test_key")
    logger.log_verification(manifest, verified)

    print("? Complete workflow executed")
    print(f"  - Traces captured: {len(manifest.trace)}")
    print("  - Manifest generated with signature")
    print(f"  - Verification: {'PASSED' if verified else 'FAILED'}")
    print("  - Logs written to: logs/calibration/workflow_*.log")
    print()
```

```
def example_create_trace_files():
    """Example: Create trace example files"""
    print("=" * 70)
    print("Example 8: Create Trace Example Files")
    print("=" * 70)

    create_trace_example(output_dir="trace_examples")

    print("? Trace examples created in: trace_examples/")
    print("  - example_traces.json")
    print()


if __name__ == "__main__":
    print("\n")
    print("*" * 70)
    print("AUDIT TRAIL SYSTEM - EXAMPLES")
    print("*" * 70)
    print("\n")

    manifest = example_basic_manifest_generation()
    example_manifest_verification(manifest)
    example_score_reconstruction(manifest)
    example_determinism_validation()
    example_structured_logging()
    example_trace_generation()
    example_complete_workflow()
    example_create_trace_files()

    print("*" * 70)
    print("ALL EXAMPLES COMPLETED SUCCESSFULLY")
    print("*" * 70)
    print()
```

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

```python
"""
Budget & Monotonicity Contract (BMC) - Implementation
"""
from typing import List, Dict, Set

class BudgetMonotonicityContract:
    @staticmethod
    def solve_knapsack(items: Dict[str, float], budget: float) -> Set[str]:
        """
        Solves a knapsack-like problem (selecting items within budget).
        To ensure monotonicity (S*(B1) ? S*(B2)), we use a greedy approach based on
cost/benefit
        or simply cost if benefit is uniform.
        Here we assume we want to maximize count of items, so we pick cheapest first.
        """
        sorted_items = sorted(items.items(), key=lambda x: x[1]) # Sort by cost
ascending

        selected = set()
        current_cost = 0.0

        for item_id, cost in sorted_items:
            if current_cost + cost <= budget:
                selected.add(item_id)
                current_cost += cost
            else:
                break

        return selected

    @staticmethod
    def verify_monotonicity(items: Dict[str, float], budgets: List[float]) -> bool:
        """
        Verifies S*(B1) ? S*(B2) for B1 < B2.
        """
        sorted_budgets = sorted(budgets)
        prev_solution = None

        for b in sorted_budgets:
            solution = BudgetMonotonicityContract.solve_knapsack(items, b)
            if prev_solution is not None:
                if not prev_solution.issubset(solution):
                    return False
            prev_solution = solution

        return True
```

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

```python
#!/usr/bin/env python3
"""
Compute and update SHA-256 hashes for all V3 executor contracts.

This script:
1. Finds all Q{nnn}.v3.json files
2. Computes SHA-256 hash of file content (excluding contract_hash field)
3. Updates contract_hash field in-place
4. Validates hash after update

Usage:
    python scripts/compute_contract_hashes.py [--dry-run] [--verify-only]
"""

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


def compute_contract_hash(contract_data: dict[str, Any]) -> str:
    """Compute SHA-256 hash of contract, excluding contract_hash field.

    Args:
        contract_data: Contract dictionary

    Returns:
        64-character lowercase hex SHA-256 hash
    """
    # Create copy and remove contract_hash to avoid circular dependency
    contract_copy = contract_data.copy()
    if "identity" in contract_copy and "contract_hash" in contract_copy["identity"]:
        contract_copy["identity"] = contract_copy["identity"].copy()
        contract_copy["identity"]["contract_hash"] = ""

    # Serialize deterministically
    json_bytes = json.dumps(
        contract_copy,
        sort_keys=True,
        separators=(",", ":"),
        ensure_ascii=False
    ).encode("utf-8")

    # Compute SHA-256
    return hashlib.sha256(json_bytes).hexdigest()


def update_contract_hash(contract_path: Path, dry_run: bool = False) -> dict[str, Any]:
    """Update contract_hash field in contract file.
    Args:
        contract_path: Path to contract JSON file
        dry_run: If True, don't write changes

    Returns:
        dict with keys: success, old_hash, new_hash, message
    """
    try:
        # Load contract
        with open(contract_path, "r", encoding="utf-8") as f:
            contract = json.load(f)

        # Validate structure
        if "identity" not in contract:
            return {
                "success": False,
                "message": "Missing 'identity' field",
                "old_hash": None,
                "new_hash": None
            }

        # Get old hash
        old_hash = contract["identity"].get("contract_hash", "")

        # Compute new hash
        new_hash = compute_contract_hash(contract)

        # Check if update needed
        if old_hash == new_hash:
            return {
                "success": True,
                "message": "Hash already correct",
                "old_hash": old_hash,
                "new_hash": new_hash,
                "updated": False
            }

        # Update hash
        contract["identity"]["contract_hash"] = new_hash

        if not dry_run:
            # Write back to file
            with open(contract_path, "w", encoding="utf-8") as f:
                json.dump(contract, f, indent=2, ensure_ascii=False)
                f.write("\n")  # Add trailing newline

        return {
            "success": True,
              "message": "Hash updated successfully" if not dry_run else "Would update
hash",
            "old_hash": old_hash,
            "new_hash": new_hash,
            "updated": True
        }
    except json.JSONDecodeError as e:
        return {
            "success": False,
            "message": f"JSON decode error: {e}",
            "old_hash": None,
            "new_hash": None
        }
    except Exception as e:
        return {
            "success": False,
            "message": f"Error: {e}",
            "old_hash": None,
            "new_hash": None
        }


def verify_contract_hash(contract_path: Path) -> dict[str, Any]:
    """Verify contract hash is correct.

    Args:
        contract_path: Path to contract JSON file

    Returns:
        dict with keys: valid, stored_hash, computed_hash, message
    """
    try:
        with open(contract_path, "r", encoding="utf-8") as f:
            contract = json.load(f)

        if "identity" not in contract or "contract_hash" not in contract["identity"]:
            return {
                "valid": False,
                "message": "Missing contract_hash field",
                "stored_hash": None,
                "computed_hash": None
            }

        stored_hash = contract["identity"]["contract_hash"]
        computed_hash = compute_contract_hash(contract)

        valid = stored_hash == computed_hash

        return {
            "valid": valid,
            "message": "Hash valid" if valid else "Hash mismatch",
            "stored_hash": stored_hash,
            "computed_hash": computed_hash
        }

    except Exception as e:
        return {
            "valid": False,
            "message": f"Error: {e}",
            "stored_hash": None,
            "computed_hash": None
        }


def main():
    parser = argparse.ArgumentParser(
        description="Compute and update SHA-256 hashes for V3 executor contracts"
    )
    parser.add_argument(
        "--dry-run",
        action="store_true",
        help="Show what would be updated without making changes"
    )
    parser.add_argument(
        "--verify-only",
        action="store_true",
        help="Only verify hashes, don't update"
    )
    parser.add_argument(
        "--contracts-dir",
        type=Path,
        help="Path to contracts directory (default: auto-detect)"
    )

    args = parser.parse_args()

    # Find contracts directory
    if args.contracts_dir:
        contracts_dir = args.contracts_dir
    else:
        # Auto-detect from script location
        script_dir = Path(__file__).parent
        project_root = script_dir.parent
            contracts_dir = project_root / "src" / "farfan_pipeline" / "phases" /
"Phase_two" / "json_files_phase_two" / "executor_contracts" / "specialized"

    if not contracts_dir.exists():
        print(f"? Contracts directory not found: {contracts_dir}", file=sys.stderr)
        sys.exit(1)

    # Find all V3 contracts
    contract_files = sorted(contracts_dir.glob("Q*.v3.json"))

    if not contract_files:
        print(f"? No V3 contracts found in {contracts_dir}", file=sys.stderr)
        sys.exit(1)

    print(f"Found {len(contract_files)} V3 contracts")
    print()

    # Process contracts
    updated = 0
    already_correct = 0
    errors = 0
    invalid_hashes = []

    for contract_path in contract_files:
        contract_id = contract_path.stem.replace(".v3", "")

        if args.verify_only:
            # Verify mode
            result = verify_contract_hash(contract_path)
            if result["valid"]:
                already_correct += 1
                print(f"? {contract_id}: {result['message']}")
            else:
                invalid_hashes.append(contract_id)
                errors += 1
                print(f"? {contract_id}: {result['message']}")
                if result["stored_hash"] and result["computed_hash"]:
                    print(f"   Stored:   {result['stored_hash'][:16]}...")
                    print(f"   Computed: {result['computed_hash'][:16]}...")
        else:
            # Update mode
            result = update_contract_hash(contract_path, dry_run=args.dry_run)
            if result["success"]:
                if result.get("updated", False):
                    updated += 1
                    print(f"? {contract_id}: {result['message']}")
                    if result["old_hash"] and result["old_hash"] != "":
                        print(f"   Old: {result['old_hash'][:16]}...")
                    print(f"   New: {result['new_hash'][:16]}...")
                else:
                    already_correct += 1
                    print(f"? {contract_id}: {result['message']}")
            else:
                errors += 1
                print(f"? {contract_id}: {result['message']}")

    print()
    print("=" * 60)
    print("Summary:")
    print(f"  Total contracts: {len(contract_files)}")

    if args.verify_only:
        print(f"  Valid hashes: {already_correct}")
        print(f"  Invalid hashes: {errors}")
        if invalid_hashes:
            print(f"  Invalid: {', '.join(invalid_hashes[:10])}")
            if len(invalid_hashes) > 10:
                print(f"           ... and {len(invalid_hashes) - 10} more")
    else:
        print(f"  Updated: {updated}")
        print(f"  Already correct: {already_correct}")
        print(f"  Errors: {errors}")

        if args.dry_run:
            print()
            print("DRY RUN - No files were modified")

    print("=" * 60)

    # Exit code
    if errors > 0 and args.verify_only:
        sys.exit(1)
    elif errors > 0:
        sys.exit(2)
    else:
        sys.exit(0)


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

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

```python
"""
Concurrency Determinism Contract (CDC) - Implementation
"""
import hashlib
import json
import threading
import time
from typing import List, Any, Callable

class ConcurrencyDeterminismContract:
    @staticmethod
    def execute_concurrently(
        func: Callable[[Any], Any],
        inputs: List[Any],
        workers: int
    ) -> List[Any]:
        """
        Simulates concurrent execution.
        To ensure determinism, results must be sorted or indexed by input ID.
        """
        results = [None] * len(inputs)

        def worker(idx, inp):
            # Simulate work
            time.sleep(0.001)
            results[idx] = func(inp)

        threads = []
        for i, inp in enumerate(inputs):
            t = threading.Thread(target=worker, args=(i, inp))
            threads.append(t)
            t.start()

            if len(threads) >= workers:
                for t in threads:
                    t.join()
                threads = []

        for t in threads:
            t.join()

        return results

    @staticmethod
    def verify_determinism(
        func: Callable[[Any], Any],
        inputs: List[Any]
    ) -> bool:
        """
        Verifies that 1 worker vs N workers produces hash-equal outputs.
        """
            res_seq = ConcurrencyDeterminismContract.execute_concurrently(func, inputs,
workers=1)
            res_conc = ConcurrencyDeterminismContract.execute_concurrently(func, inputs, workers=4)

            hash1 = hashlib.blake2b(json.dumps(res_seq, sort_keys=True).encode()).hexdigest()
            hash2 = hashlib.blake2b(json.dumps(res_conc, sort_keys=True).encode()).hexdigest()

        return hash1 == hash2
```

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

```python
from __future__ import annotations

import json
from dataclasses import is_dataclass, fields, FrozenInstanceError
from typing import Any
from collections.abc import Mapping as ABCMapping
from types import MappingProxyType
from collections.abc import Sequence as ABCSequence  # optional, for type checks

class ContextImmutabilityContract:
    @staticmethod
    def _to_plain(obj: Any) -> Any:
        """
            Convert dataclasses + immutable containers (MappingProxyType, tuples,
frozensets)
            into plain Python structures without deepcopy(). This avoids mappingproxy
pickling.
        """
        if is_dataclass(obj):
            return {f.name: ContextImmutabilityContract._to_plain(getattr(obj, f.name))
for f in fields(obj)}
        if isinstance(obj, (MappingProxyType, ABCMapping)):
            return {k: ContextImmutabilityContract._to_plain(v) for k, v in obj.items()}
        if isinstance(obj, (tuple, list, set, frozenset)):
            return [ContextImmutabilityContract._to_plain(v) for v in obj]
        return obj

    @staticmethod
    def _to_canonical(obj: Any) -> Any:
            # Sort mapping keys deterministically; lists already deterministic after
_to_plain
        if isinstance(obj, dict):
                return {k: ContextImmutabilityContract._to_canonical(obj[k]) for k in
sorted(obj.keys())}
        if isinstance(obj, list):
            return [ContextImmutabilityContract._to_canonical(v) for v in obj]
        return obj

    @staticmethod
    def canonical_digest(ctx: Any) -> str:
        # Build plain JSON-safe object without deepcopy(), then canonicalize & hash.
        plain = ContextImmutabilityContract._to_plain(ctx)
        canon = ContextImmutabilityContract._to_canonical(plain)
        s = json.dumps(canon, sort_keys=True, ensure_ascii=False, separators=(",", ":"))
        try:
            import blake3
            return blake3.blake3(s.encode("utf-8")).hexdigest()
        except Exception:
            import hashlib
            return hashlib.sha256(s.encode("utf-8")).hexdigest()

    @staticmethod
```

```
def verify_immutability(ctx: Any) -> str:
    """
    Attempt to mutate: (1) top-level attribute, (2) deep mapping.
    Both must fail. Return canonical digest for equality comparisons.
    """
    # 1) Top-level attribute mutation must fail
    try:
        setattr(ctx, "traceability_id", "MUTATE_ME")
        raise RuntimeError("Mutation succeeded but should have failed!")
    except (FrozenInstanceError, AttributeError, TypeError):
        pass  # expected

    # 2) Deep mapping mutation must fail
    deep_map = getattr(ctx, "dnp_standards", None)
    if isinstance(deep_map, (MappingProxyType, ABCMapping)):
        try:
            deep_map["__MUTATE__"] = 1  # type: ignore[index]
            raise RuntimeError("Deep mutation succeeded but should have failed!")
        except (TypeError, AttributeError):
            pass  # expected

    # 3) Deterministic canonical digest
    return ContextImmutabilityContract.canonical_digest(ctx)
```

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

```python
#!/usr/bin/env python3
"""
Contract Update Validation & Execution Tool

This tool implements a careful, phased approach to updating executor contracts:
1. Validates method signatures match
2. Creates detailed change manifest
3. Updates contracts with hash regeneration
4. Updates questionnaire_monolith.json
5. Updates executors_methods.json

Respects the manual effort invested in contract drafting with granular validation.
"""

import json
import hashlib
import inspect
import sys
from pathlib import Path
from typing import Any, Dict, List, Set, Tuple
from datetime import datetime
import importlib.util

REPO_ROOT = Path(__file__).parent
SRC_ROOT = REPO_ROOT / "src"

class ContractUpdateValidator:
    """Validates and executes contract updates with granular checks"""

    def __init__(self):
          self.contracts_dir = SRC_ROOT / "farfan_pipeline" / "phases" / "Phase_two" /
"json_files_phase_two" / "executor_contracts" / "specialized"
             self.executors_methods_path = SRC_ROOT / "farfan_pipeline" / "phases" /
"Phase_two" / "json_files_phase_two" / "executors_methods.json"
              self.questionnaire_path = REPO_ROOT / "canonic_questionnaire_central" /
"questionnaire_monolith.json"
        self.dispensary_path = SRC_ROOT / "methods_dispensary"

        self.validation_results = {
            "phase": "validation",
            "timestamp": datetime.utcnow().isoformat(),
            "method_signature_checks": [],
            "affected_contracts": [],
            "questionnaire_updates_needed": [],
            "executors_methods_updates_needed": [],
            "errors": [],
            "warnings": [],
        }

    def validate_method_replacement(
        self,
        old_class: str,
        old_method: str,
        new_class: str,
        new_method: str
    ) -> Dict[str, Any]:
        """Validate that replacement method signature is compatible"""
        print(f"\n? Validating: {old_class}.{old_method} ? {new_class}.{new_method}")

        result = {
            "old": f"{old_class}.{old_method}",
            "new": f"{new_class}.{new_method}",
            "compatible": False,
            "old_signature": None,
            "new_signature": None,
            "notes": [],
        }

        try:
            # Try to load and inspect methods
            old_sig = self._get_method_signature(old_class, old_method)
            new_sig = self._get_method_signature(new_class, new_method)

            result["old_signature"] = old_sig
            result["new_signature"] = new_sig

            if old_sig and new_sig:
                # Check parameter compatibility
                if self._signatures_compatible(old_sig, new_sig):
                    result["compatible"] = True
                    result["notes"].append("? Signatures compatible")
                    print(f"   ? Signatures compatible")
                else:
                    result["notes"].append("??  Signatures differ - may require
adapter")
                    print(f"   ??  Signatures differ - may require adapter")
            else:
                if not old_sig:
                    result["notes"].append(f"??   Could not inspect
{old_class}.{old_method}")
                    print(f"   ??  Could not inspect {old_class}.{old_method} (may be
missing)")
                if not new_sig:
                    result["notes"].append(f"? Could not inspect
{new_class}.{new_method}")
                    print(f"   ? Could not inspect {new_class}.{new_method}")

        except Exception as e:
            result["notes"].append(f"? Validation error: {e}")
            print(f"   ? Error: {e}")

        return result

    def _get_method_signature(self, class_name: str, method_name: str) -> str | None:
        """Get method signature from dispensary"""
        try:
            # Find the file containing the class
            for py_file in self.dispensary_path.glob("*.py"):
                if py_file.name.startswith("__"):
                    continue

                # Read file and look for class
                content = py_file.read_text()
                if f"class {class_name}" in content:
                    # Try to import and inspect
                    spec = importlib.util.spec_from_file_location("module", py_file)
                    if spec and spec.loader:
                        module = importlib.util.module_from_spec(spec)
                        spec.loader.exec_module(module)

                        if hasattr(module, class_name):
                            cls = getattr(module, class_name)
                            if hasattr(cls, method_name):
                                method = getattr(cls, method_name)
                                sig = inspect.signature(method)
                                return f"{method_name}{sig}"

            return None

        except Exception as e:
            print(f"   ??  Could not inspect: {e}")
            return None

    def _signatures_compatible(self, old_sig: str, new_sig: str) -> bool:
        """Check if signatures are compatible (simplified check)"""
        # For now, we consider compatible if both exist
        # More sophisticated check would parse parameters
        return old_sig is not None and new_sig is not None

    def find_affected_contracts(
        self,
        class_name: str,
        method_name: str
    ) -> List[Path]:
        """Find all contracts that use a specific method"""
        affected = []

        print(f"\n? Finding contracts using {class_name}.{method_name}...")

        for contract_file in self.contracts_dir.glob("*.v3.json"):
            try:
                with open(contract_file) as f:
                    contract = json.load(f)

                # Check method_binding.methods array
                methods = contract.get("method_binding", {}).get("methods", [])
                for method_def in methods:
                    if (method_def.get("class_name") == class_name and
                        method_def.get("method_name") == method_name):
                        affected.append(contract_file)
                        break

            except Exception as e:
                print(f"   ??  Error reading {contract_file.name}: {e}")

        print(f"   ? Found {len(affected)} affected contracts")
        return affected

    def create_change_manifest(
        self,
        replacements: List[Tuple[str, str, str, str]]
    ) -> Dict[str, Any]:
        """Create detailed manifest of all changes to be made"""
        print("\n" + "=" * 80)
        print("CREATING CHANGE MANIFEST")
        print("=" * 80)

        manifest = {
            "created_at": datetime.utcnow().isoformat(),
            "total_replacements": len(replacements),
            "replacements": [],
        }

        for old_class, old_method, new_class, new_method in replacements:
            # Validate signatures
            validation = self.validate_method_replacement(
                old_class, old_method, new_class, new_method
            )

            # Find affected contracts
            affected_contracts = self.find_affected_contracts(old_class, old_method)

            replacement_info = {
                "source": f"{old_class}.{old_method}",
                "target": f"{new_class}.{new_method}",
                "validation": validation,
                "affected_contracts": [c.name for c in affected_contracts],
                "contract_count": len(affected_contracts),
            }

            manifest["replacements"].append(replacement_info)
            self.validation_results["method_signature_checks"].append(validation)
                  self.validation_results["affected_contracts"].extend([c.name for c in
affected_contracts])

        return manifest

    def update_single_contract(
        self,
        contract_path: Path,
        old_class: str,
        old_method: str,
        new_class: str,
        new_method: str,
        dry_run: bool = True
    ) -> Dict[str, Any]:
        """Update a single contract (with dry-run mode)"""
        result = {
            "file": contract_path.name,
            "updated": False,
            "changes": [],
            "new_hash": None,
            "error": None,
        }

        try:
            with open(contract_path) as f:
                contract = json.load(f)

            # Find and update method in methods array
            methods = contract["method_binding"]["methods"]
            method_found = False

            for method_def in methods:
                if (method_def["class_name"] == old_class and
                    method_def["method_name"] == old_method):

                    if not dry_run:
                        # Update method definition
                        old_values = method_def.copy()
                        method_def["class_name"] = new_class
                        method_def["method_name"] = new_method
                        method_def["description"] = f"{new_class}.{new_method}"

                        result["changes"].append({
                            "field": "method_binding.methods",
                            "old": old_values,
                            "new": method_def.copy(),
                        })
                    else:
                        result["changes"].append({
                            "field": "method_binding.methods",
                            "action": "would_update",
                            "from": f"{old_class}.{old_method}",
                            "to": f"{new_class}.{new_method}",
                        })

                    method_found = True
                    break

            if not method_found:
                result["error"] = "Method not found in contract"
                return result

            if not dry_run:
                # Update timestamp
                    contract["identity"]["created_at"] = datetime.utcnow().isoformat() +
"+00:00"
```

```
                result["changes"].append({
                    "field": "identity.created_at",
                    "value": contract["identity"]["created_at"],
                })

                # Regenerate hash
                temp_contract = contract.copy()
                del temp_contract["identity"]["contract_hash"]
                contract_str = json.dumps(temp_contract, sort_keys=True)
                new_hash = hashlib.sha256(contract_str.encode()).hexdigest()
                contract["identity"]["contract_hash"] = new_hash
                result["new_hash"] = new_hash
                result["changes"].append({
                    "field": "identity.contract_hash",
                    "value": new_hash,
                })

                # Save contract
                with open(contract_path, "w") as f:
                    json.dump(contract, f, indent=4)

            result["updated"] = not dry_run

        except Exception as e:
            result["error"] = str(e)

        return result

    def generate_validation_report(self, manifest: Dict[str, Any]) -> None:
        """Generate comprehensive validation report"""
        print("\n" + "=" * 80)
        print("VALIDATION REPORT")
        print("=" * 80)

        print(f"\nTotal Replacements: {manifest['total_replacements']}")

        for repl in manifest["replacements"]:
            print(f"\n{repl['source']} ? {repl['target']}")
            print(f"   Contracts affected: {repl['contract_count']}")
            print(f"   Signature compatible: {repl['validation']['compatible']}")

            if repl['validation']['notes']:
                for note in repl['validation']['notes']:
                    print(f"   {note}")

        # Save to JSON
        report_path = REPO_ROOT / "contract_update_validation_report.json"
        with open(report_path, "w") as f:
            json.dump(manifest, f, indent=2)

        print(f"\n? Validation report saved: {report_path}")


def main():
    """Main validation execution"""
    print("=" * 80)
    print("CONTRACT UPDATE VALIDATOR - PHASE 1: VALIDATION")
    print("=" * 80)

    validator = ContractUpdateValidator()

    # Define high-priority replacements from audit
    high_priority_replacements = [
             ("FinancialAuditor", "_calculate_sufficiency", "PDETMunicipalPlanAnalyzer",
"_assess_financial_sustainability"),
            ("FinancialAuditor", "_detect_allocation_gaps", "PDETMunicipalPlanAnalyzer",
"_analyze_funding_sources"),
              ("FinancialAuditor", "_match_goal_to_budget", "PDETMunicipalPlanAnalyzer",
"_extract_budget_for_pillar"),
    ]

    # Create change manifest
    manifest = validator.create_change_manifest(high_priority_replacements)

    # Generate report
    validator.generate_validation_report(manifest)

    print("\n" + "=" * 80)
    print("VALIDATION COMPLETE - NO CONTRACTS MODIFIED")
    print("=" * 80)
    print("\nNext steps:")
    print("1. Review contract_update_validation_report.json")
    print("2. Verify method signatures are compatible")
    print("3. Run with --execute flag to apply changes")

    return 0


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

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

```python
"""
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/contracts_runtime.py

```python
"""
Runtime Contract Validation using Pydantic.

This module provides runtime validators for all TypedDict contracts defined
in core_contracts.py. These validators enforce:
- Value bounds and constraints
- Required vs optional fields with strict validation
- Schema versioning for backward compatibility
- Round-trip serialization guarantees

The validators mirror the TypedDict shapes exactly but add runtime enforcement.
Use these at public API boundaries and orchestrator edges.

Version: 1.0.0
Schema Version Format: sem-{major}.{minor}
"""

from typing import Any

from pydantic import BaseModel, ConfigDict, Field, field_validator

# ============================================================================
# CONFIGURATION
# ============================================================================

class StrictModel(BaseModel):
    """Base model with strict configuration for all contract validators."""

    model_config = ConfigDict(
        extra='forbid',  # Refuse unknown fields
        validate_assignment=True,
        str_strip_whitespace=True,
        validate_default=True,
        populate_by_name=True,  # Allow both field name and alias
    )

# ============================================================================
# ANALYZER_ONE.PY CONTRACTS
# ============================================================================

class SemanticAnalyzerInputModel(StrictModel):
    """Runtime validator for SemanticAnalyzerInputContract.

    Validates:
    - text is non-empty
    - schema_version follows sem-X.Y pattern
    - segments is a list of strings
    - ontology_params is a valid dict

    Example:
        >>> model = SemanticAnalyzerInputModel(
        ...     text="El plan de desarrollo municipal...",
        ...     schema_version="sem-1.0"
        ... )
    """
    text: str = Field(min_length=1, description="Document text to analyze")
    segments: list[str] = Field(
        default_factory=list,
        description="Pre-segmented text chunks"
    )
    ontology_params: dict[str, Any] = Field(
        default_factory=dict,
        description="Domain-specific ontology parameters"
    )
    schema_version: str = Field(
        default="sem-1.0",
        pattern=r"^sem-\d+\.\d+$",
        description="Contract schema version"
    )

    @field_validator('text')
    @classmethod
    def text_not_empty(cls, v: str) -> str:
        """Ensure text is not just whitespace."""
        if not v or not v.strip():
            raise ValueError("text must contain non-whitespace characters")
        return v

class SemanticAnalyzerOutputModel(StrictModel):
    """Runtime validator for SemanticAnalyzerOutputContract."""
    semantic_cube: dict[str, Any] = Field(description="Semantic analysis results")
    coherence_score: float = Field(ge=0.0, le=1.0, description="Coherence metric")
    complexity_score: float = Field(ge=0.0, description="Complexity metric")
    domain_classification: dict[str, float] = Field(
        description="Domain probability distribution"
    )
    schema_version: str = Field(
        default="sem-1.0",
        pattern=r"^sem-\d+\.\d+$"
    )

    @field_validator('domain_classification')
    @classmethod
    def validate_probabilities(cls, v: dict[str, float]) -> dict[str, float]:
        """Ensure all domain probabilities are in [0, 1]."""
        for domain, prob in v.items():
            if not (0.0 <= prob <= 1.0):
                raise ValueError(f"Probability for {domain} must be in [0, 1], got
{prob}")
        return v

# ============================================================================
# DERECK_BEACH.PY CONTRACTS
# ============================================================================

class CDAFFrameworkInputModel(StrictModel):
    """Runtime validator for CDAFFrameworkInputContract."""
    document_text: str = Field(min_length=1, description="Document to analyze")
    plan_metadata: dict[str, Any] = Field(
        default_factory=dict,
        description="Metadata about the plan"
    )
    config: dict[str, Any] = Field(
        default_factory=dict,
        description="Framework configuration"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class CDAFFrameworkOutputModel(StrictModel):
    """Runtime validator for CDAFFrameworkOutputContract."""
    causal_mechanisms: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Identified causal mechanisms"
    )
    evidential_tests: dict[str, Any] = Field(
        default_factory=dict,
        description="Statistical test results"
    )
    bayesian_inference: dict[str, Any] = Field(
        default_factory=dict,
        description="Bayesian analysis results"
    )
    audit_results: dict[str, Any] = Field(
        default_factory=dict,
        description="Audit findings and recommendations"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
# ============================================================================

class PDETAnalyzerInputModel(StrictModel):
    """Runtime validator for PDETAnalyzerInputContract."""
    document_content: str = Field(min_length=1, description="Document content")
    extract_tables: bool = Field(default=True, description="Whether to extract tables")
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class PDETAnalyzerOutputModel(StrictModel):
    """Runtime validator for PDETAnalyzerOutputContract."""
    extracted_tables: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Extracted financial tables"
    )
    financial_indicators: dict[str, float] = Field(
        default_factory=dict,
        description="Calculated financial metrics"
    )
    viability_score: float = Field(
        ge=0.0, le=1.0,
        description="Overall viability score"
    )
    quality_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Quality assessment scores"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# TEORIA_CAMBIO.PY CONTRACTS
# ============================================================================

class TeoriaCambioInputModel(StrictModel):
    """Runtime validator for TeoriaCambioInputContract."""
    document_text: str = Field(min_length=1, description="Document to analyze")
    strategic_goals: list[str] = Field(
        default_factory=list,
        description="Identified strategic goals"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class TeoriaCambioOutputModel(StrictModel):
    """Runtime validator for TeoriaCambioOutputContract."""
    causal_dag: dict[str, Any] = Field(
        default_factory=dict,
        description="Causal directed acyclic graph"
    )
    validation_results: dict[str, Any] = Field(
        default_factory=dict,
        description="Model validation results"
    )
    monte_carlo_results: dict[str, Any] | None = Field(
        default=None,
        description="Monte Carlo simulation results"
    )
    graph_visualizations: list[dict[str, Any]] | None = Field(
        default=None,
        description="Graph visualization data"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# CONTRADICTION_DETECCION.PY CONTRACTS
# ============================================================================

class ContradictionDetectorInputModel(StrictModel):
    """Runtime validator for ContradictionDetectorInputContract."""
    text: str = Field(min_length=1, description="Text to analyze for contradictions")
    plan_name: str = Field(min_length=1, description="Name of the plan")
    dimension: str | None = Field(
        default=None,
        description="PolicyDimension enum value"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class ContradictionDetectorOutputModel(StrictModel):
    """Runtime validator for ContradictionDetectorOutputContract."""
    contradictions: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Detected contradictions"
    )
    confidence_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Confidence in each detection"
    )
    temporal_conflicts: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Temporal inconsistencies"
    )
    severity_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Severity ratings"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# EMBEDDING_POLICY.PY CONTRACTS
# ============================================================================

class EmbeddingPolicyInputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyInputContract."""
    text: str = Field(min_length=1, description="Text to embed")
    dimensions: list[str] = Field(
        default_factory=list,
        description="Policy dimensions to analyze"
    )
    embedding_model_config: dict[str, Any] = Field(
        default_factory=dict,
        description="Embedding model configuration",
        alias="model_config"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class EmbeddingPolicyOutputModel(StrictModel):
    """Runtime validator for EmbeddingPolicyOutputContract."""
    embeddings: list[list[float]] = Field(
        default_factory=list,
        description="Generated embeddings"
    )
    similarity_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Similarity metrics"
    )
    bayesian_evaluation: dict[str, Any] = Field(
        default_factory=dict,
        description="Bayesian evaluation results"
    )
    policy_metrics: dict[str, float] = Field(
        default_factory=dict,
        description="Policy-specific metrics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# ============================================================================

class SemanticChunkingInputModel(StrictModel):
    """Runtime validator for SemanticChunkingInputContract."""
    text: str = Field(min_length=1, description="Text to chunk")
    preserve_structure: bool = Field(
        default=True,
        description="Whether to preserve document structure"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class SemanticChunkingOutputModel(StrictModel):
    """Runtime validator for SemanticChunkingOutputContract."""
    chunks: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Semantic chunks"
    )
    causal_dimensions: dict[str, dict[str, Any]] = Field(
        default_factory=dict,
        description="Causal dimension analysis"
    )
    key_excerpts: dict[str, list[str]] = Field(
        default_factory=dict,
        description="Key excerpts by category"
    )
    summary: dict[str, Any] = Field(
        default_factory=dict,
        description="Summary statistics"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# POLICY_PROCESSOR.PY CONTRACTS
# ============================================================================

class PolicyProcessorInputModel(StrictModel):
    """Runtime validator for PolicyProcessorInputContract."""
    data: Any = Field(description="Raw data to process")
    text: str = Field(min_length=1, description="Text content")
    sentences: list[str] = Field(
        default_factory=list,
        description="Pre-segmented sentences"
    )
    tables: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Extracted tables"
    )
    config: dict[str, Any] = Field(default_factory=dict, description="Configuration")
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

class PolicyProcessorOutputModel(StrictModel):
    """Runtime validator for PolicyProcessorOutputContract."""
    processed_data: dict[str, Any] = Field(
        default_factory=dict,
        description="Processed results"
    )
    evidence_bundles: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Evidence bundles"
    )
    bayesian_scores: dict[str, float] = Field(
        default_factory=dict,
        description="Bayesian scores"
    )
    matched_patterns: list[dict[str, Any]] = Field(
        default_factory=list,
        description="Matched patterns"
    )
    schema_version: str = Field(default="sem-1.0", pattern=r"^sem-\d+\.\d+$")

# ============================================================================
# EXPORTS
# ============================================================================

__all__ = [
    # Base
    'StrictModel',

    # Analyzer_one
    'SemanticAnalyzerInputModel',
    'SemanticAnalyzerOutputModel',

    # derek_beach
    'CDAFFrameworkInputModel',
    'CDAFFrameworkOutputModel',

    # financiero_viabilidad_tablas
    'PDETAnalyzerInputModel',
    'PDETAnalyzerOutputModel',

    # teoria_cambio
    'TeoriaCambioInputModel',
    'TeoriaCambioOutputModel',

    # contradiction_deteccion
    'ContradictionDetectorInputModel',
    'ContradictionDetectorOutputModel',
    # embedding_policy
    'EmbeddingPolicyInputModel',
    'EmbeddingPolicyOutputModel',

    # semantic_chunking_policy
    'SemanticChunkingInputModel',
    'SemanticChunkingOutputModel',

    # policy_processor
    'PolicyProcessorInputModel',
    'PolicyProcessorOutputModel',
]
```

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

```python
"""
Core Module Contracts - Type-safe API boundaries for pure library modules.

This module defines InputContract and OutputContract TypedDicts for each
core module to establish clear API boundaries and enable dependency injection.

Architectural Principles:
- Core modules receive all data via InputContract parameters
- Core modules return data via OutputContract structures
- No I/O operations within core modules
- All I/O happens in orchestrator/factory.py
- Type-safe contracts with strict typing

Version: 1.1.0
Schema Version: sem-1.0 (initial stable release)
Status: Active - Runtime validation available in contracts_runtime.py
"""

from typing import Any, TypedDict

try:
    from typing import NotRequired  # Python 3.11+
except ImportError:
    from typing_extensions import NotRequired  # Python 3.9-3.10

# ============================================================================
# ANALYZER_ONE.PY CONTRACTS
# ============================================================================

class SemanticAnalyzerInputContract(TypedDict):
    """Input contract for SemanticAnalyzer methods.

    Example:
        {
            "text": "El plan de desarrollo municipal...",
            "segments": ["Segment 1", "Segment 2"],
            "ontology_params": {"domain": "municipal"}
        }
    """
    text: str
    segments: NotRequired[list[str]]
    ontology_params: NotRequired[dict[str, Any]]

class SemanticAnalyzerOutputContract(TypedDict):
    """Output contract for SemanticAnalyzer methods."""
    semantic_cube: dict[str, Any]
    coherence_score: float
    complexity_score: float
    domain_classification: dict[str, float]

# ============================================================================
# DERECK_BEACH.PY CONTRACTS
# ============================================================================

class CDAFFrameworkInputContract(TypedDict):
    """Input contract for CDAFFramework (Causal Deconstruction Audit Framework)."""
    document_text: str
    plan_metadata: dict[str, Any]
    config: NotRequired[dict[str, Any]]

class CDAFFrameworkOutputContract(TypedDict):
    """Output contract for CDAFFramework."""
    causal_mechanisms: list[dict[str, Any]]
    evidential_tests: dict[str, Any]
    bayesian_inference: dict[str, Any]
    audit_results: dict[str, Any]

# ============================================================================
# FINANCIERO_VIABILIDAD_TABLAS.PY CONTRACTS
# ============================================================================

class PDETAnalyzerInputContract(TypedDict):
      """Input  contract  for  PDET  (Programas  de  Desarrollo  con  Enfoque  Territorial)
Analyzer."""
    document_content: str
    extract_tables: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class PDETAnalyzerOutputContract(TypedDict):
    """Output contract for PDET Analyzer."""
    extracted_tables: list[dict[str, Any]]
    financial_indicators: dict[str, float]
    viability_score: float
    quality_scores: dict[str, float]

# ============================================================================
# TEORIA_CAMBIO.PY CONTRACTS
# ============================================================================

class TeoriaCambioInputContract(TypedDict):
    """Input contract for Theory of Change analysis."""
    document_text: str
    strategic_goals: NotRequired[list[str]]
    config: NotRequired[dict[str, Any]]

class TeoriaCambioOutputContract(TypedDict):
    """Output contract for Theory of Change analysis."""
    causal_dag: dict[str, Any]
    validation_results: dict[str, Any]
    monte_carlo_results: NotRequired[dict[str, Any]]
    graph_visualizations: NotRequired[list[dict[str, Any]]]

# ============================================================================
# CONTRADICTION_DETECCION.PY CONTRACTS
# ============================================================================
```

```python
class ContradictionDetectorInputContract(TypedDict):
    """Input contract for PolicyContradictionDetector."""
    text: str
    plan_name: str
    dimension: NotRequired[str]  # PolicyDimension enum value
    config: NotRequired[dict[str, Any]]

class ContradictionDetectorOutputContract(TypedDict):
    """Output contract for PolicyContradictionDetector."""
    contradictions: list[dict[str, Any]]
    confidence_scores: dict[str, float]
    temporal_conflicts: list[dict[str, Any]]
    severity_scores: dict[str, float]

# ============================================================================
# EMBEDDING_POLICY.PY CONTRACTS
# ============================================================================

class EmbeddingPolicyInputContract(TypedDict):
    """Input contract for embedding-based policy analysis."""
    text: str
    dimensions: NotRequired[list[str]]
    model_config: NotRequired[dict[str, Any]]

class EmbeddingPolicyOutputContract(TypedDict):
    """Output contract for embedding policy analysis."""
    embeddings: list[list[float]]
    similarity_scores: dict[str, float]
    bayesian_evaluation: dict[str, Any]
    policy_metrics: dict[str, float]

# ============================================================================
# SEMANTIC_CHUNKING_POLICY.PY CONTRACTS
# ============================================================================

class SemanticChunkingInputContract(TypedDict):
    """Input contract for semantic chunking and policy document analysis."""
    text: str
    preserve_structure: NotRequired[bool]
    config: NotRequired[dict[str, Any]]

class SemanticChunkingOutputContract(TypedDict):
    """Output contract for semantic chunking."""
    chunks: list[dict[str, Any]]
    causal_dimensions: dict[str, dict[str, Any]]
    key_excerpts: dict[str, list[str]]
    summary: dict[str, Any]

# ============================================================================
# POLICY_PROCESSOR.PY CONTRACTS
# ============================================================================

class PolicyProcessorInputContract(TypedDict):
    """Input contract for IndustrialPolicyProcessor."""
    data: Any
    text: str
    sentences: NotRequired[list[str]]
    tables: NotRequired[list[dict[str, Any]]]
    config: NotRequired[dict[str, Any]]

class PolicyProcessorOutputContract(TypedDict):
    """Output contract for IndustrialPolicyProcessor."""
    processed_data: dict[str, Any]
    evidence_bundles: list[dict[str, Any]]
    bayesian_scores: dict[str, float]
    matched_patterns: list[dict[str, Any]]

# ============================================================================
# SHARED DATA STRUCTURES
# ============================================================================

class DocumentData(TypedDict):
    """Standard document data structure from farfan_core.core.orchestrator.

    This is what the orchestrator/factory provides to core modules.
    """
    raw_text: str
    sentences: list[str]
    tables: list[dict[str, Any]]
    metadata: dict[str, Any]

__all__ = [
    # Analyzer_one
    'SemanticAnalyzerInputContract',
    'SemanticAnalyzerOutputContract',

    # derek_beach
    'CDAFFrameworkInputContract',
    'CDAFFrameworkOutputContract',

    # financiero_viabilidad_tablas
    'PDETAnalyzerInputContract',
    'PDETAnalyzerOutputContract',

    # teoria_cambio
    'TeoriaCambioInputContract',
    'TeoriaCambioOutputContract',

    # contradiction_deteccion
    'ContradictionDetectorInputContract',
    'ContradictionDetectorOutputContract',

    # embedding_policy
    'EmbeddingPolicyInputContract',
    'EmbeddingPolicyOutputContract',

    # semantic_chunking_policy
    'SemanticChunkingInputContract',
    'SemanticChunkingOutputContract',

    # policy_processor
    'PolicyProcessorInputContract',
    'PolicyProcessorOutputContract',

    # Shared
    'DocumentData',
]
```