```
src/farfan_pipeline/phases/Phase_two/executor_instrumentation_mixin.py

"""
Executor Instrumentation Mixin for Calibration Integration.

This mixin adds calibration instrumentation to all D[1-6]Q[1-5] executors,
capturing runtime metrics and retrieving quality scores from the calibration system.

Usage:
                            class       D3_Q2_TargetProportionalityAnalyzer(BaseExecutor,
ExecutorInstrumentationMixin):
        def execute(self, context):
            # Instrumentation is automatically applied via wrapper
            ...
"""

from __future__ import annotations

import time
from typing import Any, Dict, Optional

from executor_calibration_integration import (
    instrument_executor,
    get_executor_config,
    CalibrationResult,
)


class ExecutorInstrumentationMixin:
    """
    Mixin to add calibration instrumentation to executors.

    This mixin provides methods to:
    1. Instrument executor execution with calibration calls
    2. Capture runtime metrics (time, memory)
    3. Retrieve quality scores from calibration system
    4. Store calibration results for reporting

    The mixin ensures NO hardcoded calibration values in executor code.
    All quality scores are loaded from external calibration files.
    """

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        super().__init__(*args, **kwargs)
        self._calibration_result: Optional[CalibrationResult] = None
        self._execution_start_time: float = 0.0
        self._execution_start_memory: float = 0.0

    def _start_calibration_tracking(self) -> None:
        """Start tracking execution metrics for calibration."""
        self._execution_start_time = time.perf_counter()

                            if   hasattr(self,   '_profiler')   and   self._profiler   and
self._profiler.memory_tracking:
```

```
            self._execution_start_memory = self._profiler._get_memory_usage_mb()
        else:
            self._execution_start_memory = 0.0

    def _stop_calibration_tracking(self, context: Dict[str, Any]) -> CalibrationResult:
        """
        Stop tracking and instrument executor with calibration call.

        Args:
            context: Execution context

        Returns:
            CalibrationResult with quality scores and metrics
        """
        runtime_ms = (time.perf_counter() - self._execution_start_time) * 1000

        memory_mb = 0.0
                            if   hasattr(self,   '_profiler')   and   self._profiler   and
self._profiler.memory_tracking:
                                    memory_mb   =   self._profiler._get_memory_usage_mb()   -
self._execution_start_memory

           methods_executed = len(self.execution_log) if hasattr(self, 'execution_log')
else 0
        methods_succeeded = sum(
               1 for log_entry in (self.execution_log if hasattr(self, 'execution_log')
else [])
            if log_entry.get('success', False)
        )

        calibration_result = instrument_executor(
            executor_id=self.executor_id,
            context=context,
            runtime_ms=runtime_ms,
            memory_mb=memory_mb,
            methods_executed=methods_executed,
            methods_succeeded=methods_succeeded
        )

        self._calibration_result = calibration_result
        return calibration_result

    def execute_with_calibration(self, context: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute with automatic calibration instrumentation.

        This wraps the execute() method to add calibration calls before
        and after execution. Quality scores are retrieved from the
        calibration system and attached to the result.

        Args:
            context: Execution context

        Returns:
```

```
        Result dict with raw_evidence and calibration metadata
    """
    self._start_calibration_tracking()

    try:
        result = self.execute(context)

        calibration_result = self._stop_calibration_tracking(context)

        if not isinstance(result, dict):
            result = {"raw_evidence": result}

        result["calibration_metadata"] = {
            "quality_score": calibration_result.quality_score,
            "layer_scores": calibration_result.layer_scores,
            "layers_used": calibration_result.layers_used,
            "aggregation_method": calibration_result.aggregation_method,
            "runtime_ms": calibration_result.metrics.runtime_ms,
            "memory_mb": calibration_result.metrics.memory_mb,
            "methods_executed": calibration_result.metrics.methods_executed,
            "methods_succeeded": calibration_result.metrics.methods_succeeded,
        }

        return result

    except Exception as e:
        calibration_result = self._stop_calibration_tracking(context)

        raise

def get_calibration_result(self) -> Optional[CalibrationResult]:
    """Get the most recent calibration result."""
    return self._calibration_result

def get_executor_runtime_config(self) -> Dict[str, Any]:
    """
    Get runtime configuration for this executor.

    This loads HOW parameters (timeout, retry, etc.) from:
    1. CLI arguments
    2. Environment variables
    3. Environment file
    4. Executor config file
    5. Conservative defaults

    Returns:
        Runtime configuration dict
    """
    parts = self.executor_id.split("_")
    if len(parts) < 2:
        return {}

    dimension = parts[0]
    question = parts[1]
```

```
        return get_executor_config(self.executor_id, dimension, question)


__all__ = ["ExecutorInstrumentationMixin"]
```

src/farfan_pipeline/phases/Phase_two/executor_profiler.py

"""Executor performance profiling framework with regression detection and dispensary analytics.

This module provides comprehensive profiling for executor performance including:
- Per-executor timing, memory, and serialization metrics
- Method call tracking with granular statistics
- Baseline comparison for regression detection
- Performance report generation identifying bottlenecks
- Integration with BaseExecutor for automatic capture
- **METHOD DISPENSARY PATTERN AWARENESS** for tracking monolith reuse

Architecture:
- ExecutorMetrics: Per-executor performance data
- MethodCallMetrics: Per-method call statistics
- ExecutorProfiler: Main profiler with baseline management + dispensary analytics
- ProfilerContext: Context manager for automatic profiling
- PerformanceReport: Structured report with bottleneck analysis

METHOD DISPENSARY INTEGRATION:
==============================
This profiler is aware of the factory's method dispensary pattern where:
- 30 executors orchestrate methods from ~20 monolith classes
- Methods are called via MethodExecutor.execute(class_name, method_name, **payload)
- Same methods are PARTIALLY reused across different executors
- Dispensary classes: PDETMunicipalPlanAnalyzer (52+ methods), CausalExtractor (28), etc.

The profiler tracks:
1. Which dispensary classes are used by each executor
2. Method reuse patterns across executors
3. Performance hotspots within dispensary classes
4. Executor-specific vs dispensary-wide bottlenecks

Usage:
    # Basic profiling
    profiler = ExecutorProfiler()
    with profiler.profile_executor("D1-Q1"):
        result = executor.execute(context)
    report = profiler.generate_report()

    # With dispensary analytics
    dispensary_stats = profiler.get_dispensary_usage_stats()
    # Shows: PDETMunicipalPlanAnalyzer used by 15 executors, avg 245ms/call
"""

from __future__ import annotations

import gc
import json
import logging
import pickle
import time

```
from collections import defaultdict
from dataclasses import asdict, dataclass, field
from datetime import datetime, timezone
from pathlib import Path
from typing import Any

logger = logging.getLogger(__name__)

# Performance thresholds (loaded from canonical_method_catalogue_v2.json via calibration
system)
# These are DEFAULT values only - actual thresholds come from method_parameters.json
DEFAULT_HIGH_EXECUTION_TIME_MS = 1000
DEFAULT_HIGH_MEMORY_MB = 100
DEFAULT_HIGH_SERIALIZATION_MS = 100

# Known dispensary classes from the method dispensary pattern
KNOWN_DISPENSARY_CLASSES = {
    "PDETMunicipalPlanAnalyzer",
    "IndustrialPolicyProcessor",
    "CausalExtractor",
    "FinancialAuditor",
    "BayesianMechanismInference",
    "BayesianCounterfactualAuditor",
    "TextMiningEngine",
    "SemanticAnalyzer",
    "PerformanceAnalyzer",
    "PolicyContradictionDetector",
    "BayesianNumericalAnalyzer",
    "TemporalLogicVerifier",
    "OperationalizationAuditor",
    "PolicyAnalysisEmbedder",
    "SemanticProcessor",
    "AdvancedDAGValidator",
    "TeoriaCambio",
    "ReportingEngine",
    "HierarchicalGenerativeModel",
    "AdaptivePriorCalculator",
    "PolicyTextProcessor",
    "MechanismPartExtractor",
    "CausalInferenceSetup",
    "BeachEvidentialTest",
    "BayesFactorTable",
    "ConfigLoader",
    "CDAFFramework",
    "IndustrialGradeValidator",
    "BayesianConfidenceCalculator",
    "PDFProcessor",
}


@dataclass
class MethodCallMetrics:
    """Metrics for a single method call within an executor.
```

```
    Enhanced to track dispensary pattern usage.
    """

    class_name: str
    method_name: str
    execution_time_ms: float
    memory_delta_mb: float
    call_count: int = 1
    success: bool = True
    error: str | None = None
    timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )

    @property
    def is_dispensary_method(self) -> bool:
        """Check if this method comes from a known dispensary class."""
        return self.class_name in KNOWN_DISPENSARY_CLASSES

    @property
    def full_method_name(self) -> str:
        """Get full method name as class.method."""
        return f"{self.class_name}.{self.method_name}"

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["is_dispensary_method"] = self.is_dispensary_method
        data["full_method_name"] = self.full_method_name
        return data


@dataclass
class ExecutorMetrics:
    """Comprehensive metrics for a single executor execution.

    Enhanced with dispensary usage tracking.
    """

    executor_id: str
    execution_time_ms: float
    memory_footprint_mb: float
    memory_peak_mb: float
    serialization_time_ms: float
    serialization_size_bytes: int
    method_calls: list[MethodCallMetrics] = field(default_factory=list)
    call_count: int = 1
    success: bool = True
    error: str | None = None
    timestamp: str = field(
        default_factory=lambda: datetime.now(timezone.utc).isoformat()
    )
    metadata: dict[str, Any] = field(default_factory=dict)
```

```
    @property
    def total_method_calls(self) -> int:
        """Total number of method calls during execution."""
        return sum(m.call_count for m in self.method_calls)

    @property
    def dispensary_method_calls(self) -> int:
        """Number of calls to dispensary methods."""
        return sum(m.call_count for m in self.method_calls if m.is_dispensary_method)

    @property
    def dispensary_usage_ratio(self) -> float:
        """Ratio of dispensary calls to total calls."""
        total = self.total_method_calls
        return self.dispensary_method_calls / total if total > 0 else 0.0

    @property
    def unique_dispensaries_used(self) -> set[str]:
        """Set of unique dispensary classes used."""
        return {m.class_name for m in self.method_calls if m.is_dispensary_method}

    @property
    def average_method_time_ms(self) -> float:
        """Average method execution time."""
        if not self.method_calls:
            return 0.0
        return sum(m.execution_time_ms for m in self.method_calls) / len(
            self.method_calls
        )

    @property
    def slowest_method(self) -> MethodCallMetrics | None:
        """Identify the slowest method call."""
        if not self.method_calls:
            return None
        return max(self.method_calls, key=lambda m: m.execution_time_ms)

    @property
    def memory_intensive_method(self) -> MethodCallMetrics | None:
        """Identify the most memory-intensive method call."""
        if not self.method_calls:
            return None
        return max(self.method_calls, key=lambda m: abs(m.memory_delta_mb))

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["method_calls"] = [m.to_dict() for m in self.method_calls]
        data["total_method_calls"] = self.total_method_calls
        data["dispensary_method_calls"] = self.dispensary_method_calls
        data["dispensary_usage_ratio"] = self.dispensary_usage_ratio
        data["unique_dispensaries_used"] = list(self.unique_dispensaries_used)
        data["average_method_time_ms"] = self.average_method_time_ms
        slowest = self.slowest_method
```

```
        data["slowest_method"] = (
            f"{slowest.class_name}.{slowest.method_name}" if slowest else None
        )
        memory_intensive = self.memory_intensive_method
        data["memory_intensive_method"] = (
            f"{memory_intensive.class_name}.{memory_intensive.method_name}"
            if memory_intensive
            else None
        )
        return data


@dataclass
class PerformanceRegression:
    """Detected performance regression for an executor."""

    executor_id: str
    metric_name: str
    baseline_value: float
    current_value: float
    delta_percent: float
    severity: str
    threshold_exceeded: bool
    recommendation: str

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


@dataclass
class PerformanceReport:
    """Comprehensive performance report with bottleneck analysis.

    Enhanced with dispensary pattern analytics.
    """

    timestamp: str
    total_executors: int
    total_execution_time_ms: float
    total_memory_mb: float
    regressions: list[PerformanceRegression] = field(default_factory=list)
    bottlenecks: list[dict[str, Any]] = field(default_factory=list)
    summary: dict[str, Any] = field(default_factory=dict)
    executor_rankings: dict[str, list[str]] = field(default_factory=dict)
    dispensary_analytics: dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> dict[str, Any]:
        """Convert to dictionary for serialization."""
        data = asdict(self)
        data["regressions"] = [r.to_dict() for r in self.regressions]
        return data
```

```
class ExecutorProfiler:
    """Performance profiler with baseline management and regression detection.

    Tracks per-executor metrics including timing, memory, serialization overhead,
    and method call counts. Supports baseline comparison for regression detection
    and generates comprehensive performance reports.

    ENHANCED: Tracks method dispensary pattern usage for monolith reuse analysis.
    """

    def __init__(
        self,
        baseline_path: Path | str | None = None,
        auto_save_baseline: bool = False,
        memory_tracking: bool = True,
        track_dispensary_usage: bool = True,
        performance_thresholds: dict[str, float] | None = None,
    ) -> None:
        """Initialize the profiler.

        Args:
            baseline_path: Path to baseline metrics file (JSON)
            auto_save_baseline: Automatically update baseline after each run
            memory_tracking: Enable memory tracking (adds overhead)
            track_dispensary_usage: Track dispensary class usage patterns
            performance_thresholds: Performance thresholds from canonical config
                                    (execution_time_ms, memory_mb, serialization_ms)
                                    If None, uses defaults from method_parameters.json
        """
        self.baseline_path = Path(baseline_path) if baseline_path else None
        self.auto_save_baseline = auto_save_baseline
        self.memory_tracking = memory_tracking
        self.track_dispensary_usage = track_dispensary_usage

        # Load thresholds from canonical config or use defaults
        self.thresholds = performance_thresholds or self._load_default_thresholds()

        self.metrics: dict[str, list[ExecutorMetrics]] = defaultdict(list)
        self.baseline_metrics: dict[str, ExecutorMetrics] = {}
        self.regressions: list[PerformanceRegression] = []

        # Dispensary usage tracking
        self.dispensary_call_counts: dict[str, int] = defaultdict(int)
        self.dispensary_execution_times: dict[str, list[float]] = defaultdict(list)
        self.executor_dispensary_usage: dict[str, set[str]] = defaultdict(set)

    def _load_default_thresholds(self) -> dict[str, float]:
        """Load default thresholds (can be overridden by canonical config)."""
        return {
            "execution_time_ms": DEFAULT_HIGH_EXECUTION_TIME_MS,
            "memory_mb": DEFAULT_HIGH_MEMORY_MB,
            "serialization_ms": DEFAULT_HIGH_SERIALIZATION_MS,
        }
```

```
        self._psutil = None
        self._psutil_process = None
        if memory_tracking:
            try:
                import psutil

                self._psutil = psutil
                self._psutil_process = psutil.Process()
            except ImportError:
                logger.warning(
                    "psutil not available, memory tracking disabled. "
                    "Install with: pip install psutil"
                )
                self.memory_tracking = False

        if self.baseline_path and self.baseline_path.exists():
            self.load_baseline(self.baseline_path)

    def _get_memory_usage_mb(self) -> float:
        """Get current memory usage in MB."""
        if not self.memory_tracking or not self._psutil_process:
            return 0.0
        try:
            return self._psutil_process.memory_info().rss / (1024 * 1024)
        except Exception as exc:
            logger.warning(f"Failed to get memory usage: {exc}")
            return 0.0

    def profile_executor(self, executor_id: str) -> ProfilerContext:
        """Create a profiling context for an executor.

        Args:
            executor_id: Unique executor identifier (e.g., "D1-Q1")

        Returns:
            ProfilerContext for use in with statement

        Example:
            with profiler.profile_executor("D1-Q1") as ctx:
                result = executor.execute(context)
                ctx.add_method_call("TextMiner", "extract", 45.2, 2.1)
        """
        return ProfilerContext(self, executor_id)

    def record_executor_metrics(
        self, executor_id: str, metrics: ExecutorMetrics
    ) -> None:
        """Record metrics for an executor execution.

        Args:
            executor_id: Unique executor identifier
            metrics: Collected metrics for the execution
        """
        self.metrics[executor_id].append(metrics)
```

```
    # Track dispensary usage
    if self.track_dispensary_usage:
        self._update_dispensary_stats(executor_id, metrics)

    if self.baseline_path and self.auto_save_baseline:
        self._update_baseline(executor_id, metrics)

def _update_dispensary_stats(
    self, executor_id: str, metrics: ExecutorMetrics
) -> None:
    """Update dispensary usage statistics.

    Args:
        executor_id: Executor identifier
        metrics: Metrics containing method calls
    """
    for method_call in metrics.method_calls:
        if method_call.is_dispensary_method:
            class_name = method_call.class_name

            # Track call counts
            self.dispensary_call_counts[class_name] += method_call.call_count

            # Track execution times
            self.dispensary_execution_times[class_name].append(
                method_call.execution_time_ms
            )

            # Track executor?dispensary usage
            self.executor_dispensary_usage[executor_id].add(class_name)

def _update_baseline(self, executor_id: str, metrics: ExecutorMetrics) -> None:
    """Update baseline with new metrics (running average).

    Args:
        executor_id: Executor identifier
        metrics: New metrics to incorporate
    """
    if executor_id not in self.baseline_metrics:
        self.baseline_metrics[executor_id] = metrics
    else:
        baseline = self.baseline_metrics[executor_id]
        baseline.execution_time_ms = (
            baseline.execution_time_ms * 0.8 + metrics.execution_time_ms * 0.2
        )
        baseline.memory_footprint_mb = (
            baseline.memory_footprint_mb * 0.8 + metrics.memory_footprint_mb * 0.2
        )
        baseline.serialization_time_ms = (
            baseline.serialization_time_ms * 0.8
            + metrics.serialization_time_ms * 0.2
        )
        baseline.call_count += 1
```

```
    def detect_regressions(
        self,
        thresholds: dict[str, float] | None = None,
    ) -> list[PerformanceRegression]:
        """Detect performance regressions against baseline.

        Args:
            thresholds: Regression thresholds for each metric
                            (default: {"execution_time_ms": 20.0, "memory_footprint_mb":
30.0})

        Returns:
            List of detected regressions
        """
        if thresholds is None:
            thresholds = {
                "execution_time_ms": 20.0,
                "memory_footprint_mb": 30.0,
                "serialization_time_ms": 50.0,
            }

        regressions: list[PerformanceRegression] = []

        for executor_id, metric_list in self.metrics.items():
            if not metric_list:
                continue

            if executor_id not in self.baseline_metrics:
                continue

            baseline = self.baseline_metrics[executor_id]
            current = metric_list[-1]

            for metric_name, threshold in thresholds.items():
                baseline_val = getattr(baseline, metric_name, 0.0)
                current_val = getattr(current, metric_name, 0.0)

                if baseline_val == 0:
                    continue

                delta_percent = ((current_val - baseline_val) / baseline_val) * 100

                if delta_percent > threshold:
                    severity = (
                        "critical" if delta_percent > threshold * 2 else "warning"
                    )
                    recommendation = self._generate_recommendation(
                        executor_id, metric_name, delta_percent, current
                    )

                    regression = PerformanceRegression(
                        executor_id=executor_id,
                        metric_name=metric_name,
```

```
                        baseline_value=baseline_val,
                        current_value=current_val,
                        delta_percent=delta_percent,
                        severity=severity,
                        threshold_exceeded=True,
                        recommendation=recommendation,
                    )
                    regressions.append(regression)

        self.regressions = regressions
        return regressions

    def _generate_recommendation(
        self,
        executor_id: str,
        metric_name: str,
        delta_percent: float,
        metrics: ExecutorMetrics | None = None,
    ) -> str:
            """Generate optimization recommendation for a regression with dispensary
awareness."""
        base_recommendations = {
            "execution_time_ms": (
                                f"Executor {executor_id} execution time increased by
{delta_percent:.1f}%. "
            ),
            "memory_footprint_mb": (
                f"Executor {executor_id} memory usage increased by {delta_percent:.1f}%.
"
            ),
            "serialization_time_ms": (
                            f"Executor {executor_id} serialization overhead increased by
{delta_percent:.1f}%. "
            ),
        }

        recommendation = base_recommendations.get(
            metric_name,
            f"Performance degradation detected in {metric_name} ({delta_percent:.1f}%)",
        )

        # Add dispensary-specific suggestions
        if metrics and self.track_dispensary_usage:
            if metric_name == "execution_time_ms" and metrics.slowest_method:
                slowest = metrics.slowest_method
                if slowest.is_dispensary_method:
                    shared_count = len(
                        [
                            eid
                                                                for eid, dispensaries in
self.executor_dispensary_usage.items()
                            if slowest.class_name in dispensaries
                        ]
                    )
```

```
                    recommendation += (
                        f"Bottleneck in dispensary method {slowest.full_method_name} "
                        f"({slowest.execution_time_ms:.1f}ms). "
                        f"Consider optimizing this method as it's shared across "
                        f"{shared_count} executors."
                    )
                else:
                        recommendation += f"Review method call sequence or optimize
{slowest.full_method_name}."
            elif metric_name == "memory_footprint_mb":
                 recommendation += "Check for memory leaks, optimize data structures, or
implement streaming."
            elif metric_name == "serialization_time_ms":
                    recommendation += "Reduce result payload size or use more efficient
serialization format."

        return recommendation

    def identify_bottlenecks(self, top_n: int = 10) -> list[dict[str, Any]]:
        """Identify top bottleneck executors requiring optimization.

        Args:
            top_n: Number of top bottlenecks to return

        Returns:
            List of bottleneck descriptors with metrics and recommendations
        """
        bottlenecks: list[dict[str, Any]] = []

        for executor_id, metric_list in self.metrics.items():
            if not metric_list:
                continue

            avg_metrics = self._compute_average_metrics(metric_list)

            bottleneck_score = (
                avg_metrics["execution_time_ms"] * 0.5
                + avg_metrics["memory_footprint_mb"] * 0.3
                + avg_metrics["serialization_time_ms"] * 0.2
            )

            bottleneck = {
                "executor_id": executor_id,
                "bottleneck_score": bottleneck_score,
                "avg_execution_time_ms": avg_metrics["execution_time_ms"],
                "avg_memory_mb": avg_metrics["memory_footprint_mb"],
                "avg_serialization_ms": avg_metrics["serialization_time_ms"],
                "total_method_calls": avg_metrics["total_method_calls"],
                "dispensary_usage_ratio": avg_metrics.get(
                    "dispensary_usage_ratio", 0.0
                ),
                "unique_dispensaries": list(
                    self.executor_dispensary_usage.get(executor_id, set())
                ),
```

```
                "slowest_method": avg_metrics["slowest_method"],
                "memory_intensive_method": avg_metrics["memory_intensive_method"],
                "recommendation": self._generate_bottleneck_recommendation(
                    executor_id, avg_metrics
                ),
            }
            bottlenecks.append(bottleneck)

        bottlenecks.sort(key=lambda x: x["bottleneck_score"], reverse=True)
        return bottlenecks[:top_n]

    def _compute_average_metrics(
        self, metric_list: list[ExecutorMetrics]
    ) -> dict[str, Any]:
        """Compute average metrics from a list of executor metrics."""
        if not metric_list:
            return {}

        return {
            "execution_time_ms": sum(m.execution_time_ms for m in metric_list)
            / len(metric_list),
            "memory_footprint_mb": sum(m.memory_footprint_mb for m in metric_list)
            / len(metric_list),
            "serialization_time_ms": sum(m.serialization_time_ms for m in metric_list)
            / len(metric_list),
            "total_method_calls": sum(m.total_method_calls for m in metric_list)
            / len(metric_list),
            "dispensary_usage_ratio": sum(m.dispensary_usage_ratio for m in metric_list)
            / len(metric_list),
            "slowest_method": (
                metric_list[-1].slowest_method.class_name
                + "."
                + metric_list[-1].slowest_method.method_name
                if metric_list[-1].slowest_method
                else None
            ),
            "memory_intensive_method": (
                metric_list[-1].memory_intensive_method.class_name
                + "."
                + metric_list[-1].memory_intensive_method.method_name
                if metric_list[-1].memory_intensive_method
                else None
            ),
        }

    def _generate_bottleneck_recommendation(
        self, _executor_id: str, avg_metrics: dict[str, Any]
    ) -> str:
             """Generate optimization recommendation for a bottleneck with dispensary
awareness."""
        recommendations = []

        if avg_metrics["execution_time_ms"] > self.thresholds["execution_time_ms"]:
            slowest = avg_metrics["slowest_method"]
```

```
            if slowest and any(
                dispensary in slowest for dispensary in KNOWN_DISPENSARY_CLASSES
            ):
                # Extract class name
                class_name = slowest.split(".")[0]
                shared_count = len(
                    [
                        eid
                        for eid, dispensaries in self.executor_dispensary_usage.items()
                        if class_name in dispensaries
                    ]
                )
                recommendations.append(
                    f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
                    f"dispensary method {slowest} shared by {shared_count} executors - "
                    f"optimization here benefits multiple executors"
                )
            else:
                recommendations.append(
                    f"High execution time ({avg_metrics['execution_time_ms']:.1f}ms): "
                    f"optimize {slowest or 'slow methods'}"
                )

        if avg_metrics["memory_footprint_mb"] > self.thresholds["memory_mb"]:
            recommendations.append(
                f"High memory usage ({avg_metrics['memory_footprint_mb']:.1f}MB): "
                f"review {avg_metrics['memory_intensive_method'] or 'data structures'}"
            )

        if avg_metrics["serialization_time_ms"] > self.thresholds["serialization_ms"]:
            recommendations.append(
                                                    f"High   serialization   overhead
({avg_metrics['serialization_time_ms']:.1f}ms): "
                "reduce payload size"
            )

        if not recommendations:
            return "Performance acceptable, monitor for regressions"

        return "; ".join(recommendations)

    def get_dispensary_usage_stats(self) -> dict[str, Any]:
        """Get comprehensive dispensary usage statistics."""
        if not self.track_dispensary_usage:
            return {
                "tracking_enabled": False,
                                "message":  "Dispensary  tracking  disabled.  Enable  with
track_dispensary_usage=True",
            }

        dispensary_stats = {}

        for dispensary_class in KNOWN_DISPENSARY_CLASSES:
            if dispensary_class not in self.dispensary_call_counts:
```

```
            continue

        call_count = self.dispensary_call_counts[dispensary_class]
        exec_times = self.dispensary_execution_times.get(dispensary_class, [])

        avg_time = sum(exec_times) / len(exec_times) if exec_times else 0.0
        total_time = sum(exec_times)

        # Find which executors use this dispensary
        using_executors = [
            eid
            for eid, dispensaries in self.executor_dispensary_usage.items()
            if dispensary_class in dispensaries
        ]

        dispensary_stats[dispensary_class] = {
            "total_calls": call_count,
            "avg_execution_time_ms": avg_time,
            "total_execution_time_ms": total_time,
            "used_by_executor_count": len(using_executors),
            "using_executors": using_executors,
            "reuse_factor": call_count / max(len(using_executors), 1),
        }

    # Sort by total execution time
    sorted_dispensaries = sorted(
        dispensary_stats.items(),
        key=lambda x: x[1]["total_execution_time_ms"],
        reverse=True,
    )

    return {
        "tracking_enabled": True,
        "total_dispensaries_used": len(dispensary_stats),
        "total_dispensary_calls": sum(self.dispensary_call_counts.values()),
        "dispensaries": dict(sorted_dispensaries),
        "hottest_dispensaries": [
            {
                "class": name,
                "total_time_ms": stats["total_execution_time_ms"],
                "avg_time_ms": stats["avg_execution_time_ms"],
                "executor_count": stats["used_by_executor_count"],
                "reuse_factor": stats["reuse_factor"],
            }
            for name, stats in sorted_dispensaries[:5]
        ],
    }

def generate_report(
    self, include_regressions: bool = True, include_bottlenecks: bool = True
) -> PerformanceReport:
    """Generate comprehensive performance report.

    Args:
```

```
    include_regressions: Include regression detection
    include_bottlenecks: Include bottleneck analysis

Returns:
    PerformanceReport with analysis and recommendations
"""
regressions = []
if include_regressions:
    regressions = self.detect_regressions()

bottlenecks = []
if include_bottlenecks:
    bottlenecks = self.identify_bottlenecks()

total_execution_time = sum(
    m.execution_time_ms for metrics in self.metrics.values() for m in metrics
)
total_memory = sum(
    m.memory_footprint_mb for metrics in self.metrics.values() for m in metrics
)

executor_rankings = {
    "slowest": self._rank_executors_by("execution_time_ms"),
    "memory_intensive": self._rank_executors_by("memory_footprint_mb"),
    "serialization_heavy": self._rank_executors_by("serialization_time_ms"),
}

summary = {
    "total_executors_profiled": len(self.metrics),
    "total_executions": sum(len(m) for m in self.metrics.values()),
    "regressions_detected": len(regressions),
    "critical_regressions": sum(
        1 for r in regressions if r.severity == "critical"
    ),
    "bottlenecks_identified": len(bottlenecks),
    "avg_execution_time_ms": total_execution_time
    / max(1, sum(len(m) for m in self.metrics.values())),
    "avg_memory_mb": total_memory
    / max(1, sum(len(m) for m in self.metrics.values())),
}

# Add dispensary analytics to report
dispensary_analytics = {}
if self.track_dispensary_usage:
    dispensary_analytics = self.get_dispensary_usage_stats()

return PerformanceReport(
    timestamp=datetime.now(timezone.utc).isoformat(),
    total_executors=len(self.metrics),
    total_execution_time_ms=total_execution_time,
    total_memory_mb=total_memory,
    regressions=regressions,
    bottlenecks=bottlenecks,
    summary=summary,
```

```
        executor_rankings=executor_rankings,
        dispensary_analytics=dispensary_analytics,
    )

def _rank_executors_by(self, metric_name: str, top_n: int = 10) -> list[str]:
    """Rank executors by a specific metric.

    Args:
        metric_name: Metric to rank by
        top_n: Number of top executors to return

    Returns:
        List of executor IDs ranked by metric
    """
    rankings = []
    for executor_id, metric_list in self.metrics.items():
        if not metric_list:
            continue
        avg_value = sum(getattr(m, metric_name, 0.0) for m in metric_list) / len(
            metric_list
        )
        rankings.append((executor_id, avg_value))

    rankings.sort(key=lambda x: x[1], reverse=True)
    return [executor_id for executor_id, _ in rankings[:top_n]]

def save_baseline(self, path: Path | str | None = None) -> None:
    """Save current metrics as baseline.

    Args:
        path: Path to save baseline (uses self.baseline_path if None)
    """
    path = Path(path) if path else self.baseline_path
    if not path:
        raise ValueError("No baseline path specified")

    path.parent.mkdir(parents=True, exist_ok=True)

    baseline_data = {
        executor_id: metrics.to_dict()
        for executor_id, metrics in self.baseline_metrics.items()
    }

    with open(path, "w", encoding="utf-8") as f:
        json.dump(baseline_data, f, indent=2)

    logger.info(f"Baseline saved to {path}")

def load_baseline(self, path: Path | str) -> None:
    """Load baseline metrics from file.

    Args:
        path: Path to baseline file
    """
```

```
    path = Path(path)
    if not path.exists():
        logger.warning(f"Baseline file not found: {path}")
        return

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

    for executor_id, data in baseline_data.items():
        method_calls = [
            MethodCallMetrics(**m) for m in data.pop("method_calls", [])
        ]
        # Remove computed properties before reconstructing
        data.pop("total_method_calls", None)
        data.pop("dispensary_method_calls", None)
        data.pop("dispensary_usage_ratio", None)
        data.pop("unique_dispensaries_used", None)
        data.pop("average_method_time_ms", None)
        data.pop("slowest_method", None)
        data.pop("memory_intensive_method", None)

        metrics = ExecutorMetrics(**data, method_calls=method_calls)
        self.baseline_metrics[executor_id] = metrics

    logger.info(
        f"Baseline loaded from {path}: {len(self.baseline_metrics)} executors"
    )

def export_report(
    self, report: PerformanceReport, path: Path | str, format: str = "json"
) -> None:
    """Export performance report to file.

    Args:
        report: Performance report to export
        path: Output path
        format: Output format ("json", "markdown", or "html")
    """
    path = Path(path)
    path.parent.mkdir(parents=True, exist_ok=True)

    if format == "json":
        with open(path, "w", encoding="utf-8") as f:
            json.dump(report.to_dict(), f, indent=2)

    elif format == "markdown":
        self._export_markdown(report, path)

    elif format == "html":
        self._export_html(report, path)

    else:
        raise ValueError(f"Unsupported format: {format}")
```

```
        logger.info(f"Report exported to {path}")

    def _export_markdown(self, report: PerformanceReport, path: Path) -> None:
        """Export report as Markdown."""
        lines = [
            "# Executor Performance Report",
            f"**Generated:** {report.timestamp}",
            "",
            "## Summary",
            f"- **Total Executors:** {report.total_executors}",
            f"- **Total Execution Time:** {report.total_execution_time_ms:.2f}ms",
            f"- **Total Memory:** {report.total_memory_mb:.2f}MB",
               f"- **Regressions Detected:** {report.summary.get('regressions_detected',
0)}",
                                                    f"-    **Bottlenecks    Identified:**
{report.summary.get('bottlenecks_identified', 0)}",
            "",
        ]

        # Dispensary analytics section
        if report.dispensary_analytics.get("tracking_enabled"):
            lines.extend(
                [
                    "## Dispensary Usage Analytics",
                    "",
                                                      f"-   **Total   Dispensaries   Used:**
{report.dispensary_analytics.get('total_dispensaries_used', 0)}",
                                                       f"-   **Total   Dispensary   Calls:**
{report.dispensary_analytics.get('total_dispensary_calls', 0)}",
                    "",
                    "### Hottest Dispensaries",
                    "",
                       "| Rank | Class | Total Time (ms) | Avg Time (ms) | Executors |
Reuse Factor |",

"|------|-------|----------------|---------------|-----------|--------------|",
                ]
            )

            for i, disp in enumerate(
                report.dispensary_analytics.get("hottest_dispensaries", []), 1
            ):
                lines.append(
                    f"| {i} | {disp['class']} | {disp['total_time_ms']:.1f} | "
                    f"{disp['avg_time_ms']:.1f} | {disp['executor_count']} | "
                    f"{disp['reuse_factor']:.1f} |"
                )
            lines.append("")

        if report.regressions:
            lines.extend(
                [
                    "## Performance Regressions",
                    "",
```

```
                    "| Executor | Metric | Baseline | Current | Delta | Severity |",
                    "|----------|--------|----------|---------|-------|----------|",
                ]
            )
            for reg in report.regressions:
                lines.append(
                    f"| {reg.executor_id} | {reg.metric_name} | "
                    f"{reg.baseline_value:.2f} | {reg.current_value:.2f} | "
                    f"{reg.delta_percent:+.1f}% | {reg.severity} |"
                )
            lines.append("")

        if report.bottlenecks:
            lines.extend(
                [
                    "## Top Bottlenecks",
                    "",
                       "| Rank | Executor | Score | Exec Time | Memory | Dispensaries |
Recommendation |",

"|------|----------|-------|-----------|--------|--------------|----------------|",
                ]
            )
            for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
                disp_count = len(bottleneck.get("unique_dispensaries", []))
                lines.append(
                    f"| {i} | {bottleneck['executor_id']} | "
                    f"{bottleneck['bottleneck_score']:.1f} | "
                    f"{bottleneck['avg_execution_time_ms']:.1f}ms | "
                    f"{bottleneck['avg_memory_mb']:.1f}MB | "
                    f"{disp_count} | "
                    f"{bottleneck['recommendation'][:60]}... |"
                )
            lines.append("")

        with open(path, "w", encoding="utf-8") as f:
            f.write("\n".join(lines))

    def _export_html(self, report: PerformanceReport, path: Path) -> None:
        """Export report as HTML."""
        html = f"""<!DOCTYPE html>
<html>
<head>
    <title>Executor Performance Report</title>
    <style>
        body {{ font-family: Arial, sans-serif; margin: 20px; }}
        h1, h2 {{ color: #333; }}
        table {{ border-collapse: collapse; width: 100%; margin: 20px 0; }}
        th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}
        th {{ background-color: #4CAF50; color: white; }}
        .critical {{ color: red; font-weight: bold; }}
        .warning {{ color: orange; font-weight: bold; }}
    </style>
</head>
```

```
<body>
    <h1>Executor Performance Report</h1>
    <p><strong>Generated:</strong> {report.timestamp}</p>

    <h2>Summary</h2>
    <ul>
        <li><strong>Total Executors:</strong> {report.total_executors}</li>
                                        <li><strong>Total     Execution     Time:</strong>
{report.total_execution_time_ms:.2f}ms</li>
        <li><strong>Total Memory:</strong> {report.total_memory_mb:.2f}MB</li>
                                            <li><strong>Regressions     Detected:</strong>
{report.summary.get('regressions_detected', 0)}</li>
                                          <li><strong>Bottlenecks     Identified:</strong>
{report.summary.get('bottlenecks_identified', 0)}</li>
    </ul>
"""

        # Add dispensary analytics
        if report.dispensary_analytics.get("tracking_enabled"):
            html += """
    <h2>Dispensary Usage Analytics</h2>
    <table>
        <tr>
            <th>Rank</th>
            <th>Dispensary Class</th>
            <th>Total Time (ms)</th>
            <th>Avg Time (ms)</th>
            <th>Executor Count</th>
            <th>Reuse Factor</th>
        </tr>
"""

            for i, disp in enumerate(
                report.dispensary_analytics.get("hottest_dispensaries", []), 1
            ):
                html += f"""
        <tr>
            <td>{i}</td>
            <td>{disp['class']}</td>
            <td>{disp['total_time_ms']:.1f}</td>
            <td>{disp['avg_time_ms']:.1f}</td>
            <td>{disp['executor_count']}</td>
            <td>{disp['reuse_factor']:.1f}</td>
        </tr>
"""
            html += "    </table>\n"

        if report.regressions:
            html += """
    <h2>Performance Regressions</h2>
    <table>
        <tr>
            <th>Executor</th>
            <th>Metric</th>
```

```
            <th>Baseline</th>
            <th>Current</th>
            <th>Delta</th>
            <th>Severity</th>
        </tr>
"""
            for reg in report.regressions:
                severity_class = reg.severity
                html += f"""
        <tr>
            <td>{reg.executor_id}</td>
            <td>{reg.metric_name}</td>
            <td>{reg.baseline_value:.2f}</td>
            <td>{reg.current_value:.2f}</td>
            <td>{reg.delta_percent:+.1f}%</td>
            <td class="{severity_class}">{reg.severity}</td>
        </tr>
"""
            html += "    </table>\n"

        if report.bottlenecks:
            html += """
    <h2>Top Bottlenecks</h2>
    <table>
        <tr>
            <th>Rank</th>
            <th>Executor</th>
            <th>Score</th>
            <th>Exec Time</th>
            <th>Memory</th>
            <th>Recommendation</th>
        </tr>
"""
            for i, bottleneck in enumerate(report.bottlenecks[:10], 1):
                html += f"""
        <tr>
            <td>{i}</td>
            <td>{bottleneck['executor_id']}</td>
            <td>{bottleneck['bottleneck_score']:.1f}</td>
            <td>{bottleneck['avg_execution_time_ms']:.1f}ms</td>
            <td>{bottleneck['avg_memory_mb']:.1f}MB</td>
            <td>{bottleneck['recommendation']}</td>
        </tr>
"""
            html += "    </table>\n"

        html += """
</body>
</html>
"""
        with open(path, "w", encoding="utf-8") as f:
            f.write(html)

    def clear_metrics(self) -> None:
```

```
        """Clear all collected metrics (but not baseline)."""
        self.metrics.clear()
        self.regressions.clear()
        self.dispensary_call_counts.clear()
        self.dispensary_execution_times.clear()
        self.executor_dispensary_usage.clear()


class ProfilerContext:
    """Context manager for automatic executor profiling.

    Automatically captures timing, memory, and serialization metrics
    when used with a 'with' statement.
    """

    def __init__(self, profiler: ExecutorProfiler, executor_id: str) -> None:
        """Initialize profiler context.

        Args:
            profiler: Parent profiler instance
            executor_id: Executor being profiled
        """
        self.profiler = profiler
        self.executor_id = executor_id
        self.start_time: float = 0.0
        self.start_memory: float = 0.0
        self.method_calls: list[MethodCallMetrics] = []
        self.result: Any = None
        self.error: str | None = None

    def __enter__(self) -> ProfilerContext:
        """Enter profiling context."""
        self.start_time = time.perf_counter()
        self.start_memory = self.profiler._get_memory_usage_mb()
        gc.collect()
        return self

    def __exit__(
        self,
        exc_type: type[BaseException] | None,
        exc_val: BaseException | None,
        exc_tb: object,
    ) -> None:
        """Exit profiling context and record metrics."""
        execution_time = (time.perf_counter() - self.start_time) * 1000
        end_memory = self.profiler._get_memory_usage_mb()
        memory_footprint = end_memory - self.start_memory
        memory_peak = max(end_memory, self.start_memory)

        serialization_time, serialization_size = self._measure_serialization()

        metrics = ExecutorMetrics(
            executor_id=self.executor_id,
            execution_time_ms=execution_time,
```

```
            memory_footprint_mb=memory_footprint,
            memory_peak_mb=memory_peak,
            serialization_time_ms=serialization_time,
            serialization_size_bytes=serialization_size,
            method_calls=self.method_calls,
            success=exc_type is None,
            error=str(exc_val) if exc_val else None,
        )

        self.profiler.record_executor_metrics(self.executor_id, metrics)

    def _measure_serialization(self) -> tuple[float, int]:
        """Measure serialization overhead for the result.

        Returns:
            Tuple of (serialization_time_ms, serialization_size_bytes)
        """
        if self.result is None:
            return 0.0, 0

        try:
            start = time.perf_counter()
            serialized = pickle.dumps(self.result, protocol=pickle.HIGHEST_PROTOCOL)
            serialization_time = (time.perf_counter() - start) * 1000
            serialization_size = len(serialized)
            return serialization_time, serialization_size
        except Exception as exc:
            logger.warning(f"Failed to measure serialization: {exc}")
            return 0.0, 0

    def add_method_call(
        self,
        class_name: str,
        method_name: str,
        execution_time_ms: float,
        memory_delta_mb: float = 0.0,
        success: bool = True,
        error: str | None = None,
    ) -> None:
        """Add a method call to the profiling context.

        Args:
            class_name: Class of the method
            method_name: Name of the method
            execution_time_ms: Execution time in milliseconds
            memory_delta_mb: Memory delta in MB
            success: Whether the call succeeded
            error: Error message if failed
        """
        metrics = MethodCallMetrics(
            class_name=class_name,
            method_name=method_name,
            execution_time_ms=execution_time_ms,
            memory_delta_mb=memory_delta_mb,
```

```
            success=success,
            error=error,
        )
        self.method_calls.append(metrics)

    def set_result(self, result: object) -> None:
        """Set the result for serialization measurement.

        Args:
            result: Execution result (can be any serializable object)
        """
        self.result = result


__all__ = [
    "ExecutorProfiler",
    "ProfilerContext",
    "ExecutorMetrics",
    "MethodCallMetrics",
    "PerformanceRegression",
    "PerformanceReport",
    "KNOWN_DISPENSARY_CLASSES",
]
```

src/farfan_pipeline/phases/Phase_two/executor_tests.py

```
"""
Test suite for executor calibration integration.

Tests verify:
1. All 30+ executors are instrumented with calibration calls
2. Calibration data (WHAT) is loaded from intrinsic_calibration.json
3. Runtime parameters (HOW) are loaded from ExecutorConfig
4. No hardcoded calibration values in executor code
5. Layer assignments are correct (role:SCORE_Q with all 8 layers)
6. Choquet integral aggregation works correctly
"""

from __future__ import annotations

import json
from pathlib import Path
from typing import Dict, Any, List

import pytest

from executor_calibration_integration import (
    CalibrationIntegration,
    CalibrationContext,
    CalibrationMetrics,
    CalibrationResult,
    instrument_executor,
    get_executor_config,
    EXECUTOR_REQUIRED_LAYERS,
    LAYER_WEIGHTS,
    INTERACTION_WEIGHTS,
)


# Test data: all 30 executor IDs
ALL_EXECUTOR_IDS = [
    "D1_Q1_QuantitativeBaselineExtractor",
    "D1_Q2_ProblemDimensioningAnalyzer",
    "D1_Q3_BudgetAllocationTracer",
    "D1_Q4_InstitutionalCapacityIdentifier",
    "D1_Q5_ScopeJustificationValidator",
    "D2_Q1_StructuredPlanningValidator",
    "D2_Q2_InterventionLogicInferencer",
    "D2_Q3_RootCauseLinkageAnalyzer",
    "D2_Q4_RiskManagementAnalyzer",
    "D2_Q5_StrategicCoherenceEvaluator",
    "D3_Q1_IndicatorQualityValidator",
    "D3_Q2_TargetProportionalityAnalyzer",
    "D3_Q3_TraceabilityValidator",
    "D3_Q4_TechnicalFeasibilityEvaluator",
    "D3_Q5_OutputOutcomeLinkageAnalyzer",
    "D4_Q1_OutcomeMetricsValidator",
    "D4_Q2_CausalChainValidator",
```

```
    "D4_Q3_AmbitionJustificationAnalyzer",
    "D4_Q4_ProblemSolvencyEvaluator",
    "D4_Q5_VerticalAlignmentValidator",
    "D5_Q1_LongTermVisionAnalyzer",
    "D5_Q2_CompositeMeasurementValidator",
    "D5_Q3_IntangibleMeasurementAnalyzer",
    "D5_Q4_SystemicRiskEvaluator",
    "D5_Q5_RealismAndSideEffectsAnalyzer",
    "D6_Q1_ExplicitTheoryBuilder",
    "D6_Q2_LogicalProportionalityValidator",
    "D6_Q3_ValidationTestingAnalyzer",
    "D6_Q4_FeedbackLoopAnalyzer",
    "D6_Q5_ContextualAdaptabilityEvaluator",
]


class TestExecutorCalibrationIntegration:
    """Test suite for calibration integration system."""

    def test_all_executors_count(self):
        """Verify we have exactly 30 executors."""
        assert len(ALL_EXECUTOR_IDS) == 30, \
            f"Expected 30 executors, found {len(ALL_EXECUTOR_IDS)}"

    def test_executor_id_format(self):
        """Verify all executor IDs follow D[1-6]_Q[1-5]_Name format."""
        import re
        pattern = re.compile(r"^D[1-6]_Q[1-5]_\w+$")

        for executor_id in ALL_EXECUTOR_IDS:
            assert pattern.match(executor_id), \
                f"Executor ID {executor_id} does not match expected format"

    def test_calibration_integration_initialization(self):
        """Test CalibrationIntegration can be initialized."""
        integration = CalibrationIntegration()

        assert integration.calibration_data is not None
        assert integration.questionnaire_data is not None
        assert isinstance(integration.calibration_data, dict)
        assert isinstance(integration.questionnaire_data, dict)

    def test_layer_requirements(self):
        """Verify all executors require the correct 8 layers."""
        expected_layers = {"@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"}
        actual_layers = set(EXECUTOR_REQUIRED_LAYERS)

        assert actual_layers == expected_layers, \
            f"Layer mismatch: expected {expected_layers}, got {actual_layers}"

    def test_layer_weights_sum(self):
        """Verify layer weights + interaction weights sum to ~1.0."""
        linear_sum = sum(LAYER_WEIGHTS.values())
        interaction_sum = sum(INTERACTION_WEIGHTS.values())
```

```
        total = linear_sum + interaction_sum

        assert abs(total - 1.0) < 0.01, \
            f"Weights should sum to 1.0, got {total}"

    def test_instrument_executor_basic(self):
        """Test basic executor instrumentation."""
        context = {
            "document_text": "Test document",
            "policy_area": "PA01",
            "unit_id": "test_unit"
        }

        result = instrument_executor(
            executor_id="D3_Q2_TargetProportionalityAnalyzer",
            context=context,
            runtime_ms=1500.0,
            memory_mb=256.0,
            methods_executed=21,
            methods_succeeded=21
        )

        assert isinstance(result, CalibrationResult)
                                                    assert      result.method_id      ==
"farfan_pipeline.core.orchestrator.executors.D3_Q2_TargetProportionalityAnalyzer"
        assert result.context.dimension == "D3"
        assert result.context.question == "Q2"
        assert result.metrics.runtime_ms == 1500.0
        assert result.metrics.memory_mb == 256.0
        assert 0.0 <= result.quality_score <= 1.0
        assert len(result.layer_scores) == 8
        assert result.aggregation_method == "choquet"

    def test_quality_score_calculation(self):
        """Test quality score calculation for various executors."""
        integration = CalibrationIntegration()

        for executor_id in ALL_EXECUTOR_IDS[:5]:  # Test first 5
            parts = executor_id.split("_")
            dimension = parts[0]
            question = parts[1]

            context = CalibrationContext(
                question=question,
                dimension=dimension,
                executor_class=executor_id
            )

            quality_score, layer_scores = integration.calculate_quality_score(
                f"farfan_pipeline.core.orchestrator.executors.{executor_id}",
                context
            )

            assert 0.0 <= quality_score <= 1.0, \
```

```
                f"Quality score {quality_score} out of range for {executor_id}"
            assert len(layer_scores) == 8, \
                f"Expected 8 layer scores, got {len(layer_scores)} for {executor_id}"

    def test_executor_config_loading(self):
        """Test loading executor-specific configuration."""
        config = get_executor_config(
            executor_id="D3_Q2_TargetProportionalityAnalyzer",
            dimension="D3",
            question="Q2"
        )

        assert config["executor_id"] == "D3_Q2_TargetProportionalityAnalyzer"
        assert config["dimension"] == "D3"
        assert config["question"] == "Q2"
        assert config["role"] == "SCORE_Q"
        assert len(config["required_layers"]) == 8
        assert "layer_weights" in config
        assert "interaction_weights" in config

    def test_no_hardcoded_calibration_values(self):
        """Verify no hardcoded calibration values in executor code."""
        executors_file = Path(__file__).parent / "executors.py"

        with open(executors_file) as f:
            content = f.read()

        # Check that calibration-related constants are NOT defined
        forbidden_patterns = [
            "QUALITY_SCORE =",
            "CALIBRATION_VALUE =",
            "BASE_SCORE =",
        ]

        for pattern in forbidden_patterns:
            assert pattern not in content, \
                f"Found hardcoded calibration pattern: {pattern}"

    def test_calibration_parametrization_separation(self):
        """Verify calibration and parametrization are properly separated."""
        # Calibration data should NOT contain runtime parameters
        integration = CalibrationIntegration()
        calibration_data = integration.calibration_data

        forbidden_runtime_keys = ["timeout_s", "retry", "temperature", "max_tokens"]

        def check_dict_recursive(d: Dict, path: str = "") -> None:
            for key, value in d.items():
                current_path = f"{path}.{key}" if path else key
                assert key not in forbidden_runtime_keys, \
                                f"Found runtime parameter {key} in calibration data at
{current_path}"
                if isinstance(value, dict):
                    check_dict_recursive(value, current_path)
```

```
    check_dict_recursive(calibration_data)

def test_executor_config_files_exist(self):
    """Verify executor config files exist for sample executors."""
    config_dir = Path(__file__).parent / "executor_configs"

    sample_executors = [
        "D1_Q1_QuantitativeBaselineExtractor",
        "D3_Q2_TargetProportionalityAnalyzer",
        "D6_Q1_ExplicitTheoryBuilder",
    ]

    for executor_id in sample_executors:
        config_file = config_dir / f"{executor_id}.json"
        assert config_file.exists(), \
            f"Config file not found: {config_file}"

        with open(config_file) as f:
            config = json.load(f)

        assert config["executor_id"] == executor_id
        assert "runtime_parameters" in config
        assert "thresholds" in config
        assert "required_layers" in config

def test_calibration_report_completeness(self):
    """Verify calibration report documents all 30 executors."""
    report_file = Path(__file__).parent / "executor_calibration_report.json"

    with open(report_file) as f:
        report = json.load(f)

    assert report["metadata"]["total_executors"] == 30
    assert len(report["executors"]) == 30

    reported_ids = {e["executor_id"] for e in report["executors"]}
    expected_ids = set(ALL_EXECUTOR_IDS)

    assert reported_ids == expected_ids, \
        f"Report missing executors: {expected_ids - reported_ids}"

def test_layer_score_retrieval(self):
    """Test retrieving individual layer scores."""
    integration = CalibrationIntegration()
    context = CalibrationContext(
        question="Q2",
        dimension="D3",
        executor_class="D3_Q2_TargetProportionalityAnalyzer"
    )

    for layer in EXECUTOR_REQUIRED_LAYERS:
        score = integration.get_layer_score(layer, context)
        assert 0.0 <= score <= 1.0, \
```

```
                f"Layer {layer} score {score} out of range"

    def test_choquet_aggregation(self):
        """Test Choquet integral aggregation."""
        integration = CalibrationIntegration()
        context = CalibrationContext(
            question="Q2",
            dimension="D3",
            executor_class="D3_Q2_TargetProportionalityAnalyzer"
        )

        quality_score, layer_scores = integration.calculate_quality_score(

"farfan_pipeline.core.orchestrator.executors.D3_Q2_TargetProportionalityAnalyzer",
            context
        )

        # Verify Choquet components
        linear_sum = sum(
            LAYER_WEIGHTS.get(layer, 0.0) * score
            for layer, score in layer_scores.items()
        )

        interaction_sum = 0.0
        for (l1, l2), weight in INTERACTION_WEIGHTS.items():
            if l1 in layer_scores and l2 in layer_scores:
                interaction_sum += weight * min(layer_scores[l1], layer_scores[l2])

        expected_score = linear_sum + interaction_sum

        assert abs(quality_score - expected_score) < 0.01, \
            f"Choquet aggregation mismatch: {quality_score} != {expected_score}"


class TestExecutorInstrumentation:
    """Test individual executor instrumentation."""

    @pytest.mark.parametrize("executor_id", ALL_EXECUTOR_IDS)
    def test_executor_instrumentation(self, executor_id: str):
        """Test instrumentation for each executor."""
        parts = executor_id.split("_")
        dimension = parts[0]
        question = parts[1]

        context = {
            "document_text": "Test document",
            "policy_area": "PA01",
            "unit_id": "test_unit"
        }

        result = instrument_executor(
            executor_id=executor_id,
            context=context,
            runtime_ms=1000.0,
```

```
            memory_mb=128.0,
            methods_executed=10,
            methods_succeeded=10
        )

        assert result.context.dimension == dimension
        assert result.context.question == question
        assert result.quality_score >= 0.0
        assert result.quality_score <= 1.0
        assert len(result.layers_used) == 8


class TestConfigurationLoading:
    """Test configuration loading from multiple sources."""

    def test_config_hierarchy_cli_env_json(self):
        """Test config loading hierarchy: CLI > ENV > JSON."""
        # This would test the actual ExecutorConfig class
        # which should load from CLI args, then ENV vars, then JSON files
        # Placeholder for actual implementation
        pass

    def test_executor_config_schema(self):
        """Test executor config files follow correct schema."""
        config_dir = Path(__file__).parent / "executor_configs"
        template_file = config_dir / "executor_config_template.json"

        if not template_file.exists():
            pytest.skip("Template file not found")

        with open(template_file) as f:
            template = json.load(f)

        required_keys = [
            "executor_id",
            "dimension",
            "question",
            "role",
            "required_layers",
            "runtime_parameters",
            "thresholds",
            "epistemic_mix",
            "contextual_params"
        ]

        for key in required_keys:
            assert key in template, f"Template missing required key: {key}"


if __name__ == "__main__":
    pytest.main([__file__, "-v"])
```

src/farfan_pipeline/phases/Phase_two/executors.py

```
from __future__ import annotations

from canonic_phases.Phase_two.base_executor_with_contract import
BaseExecutorWithContract


class D1Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q1"


class D1Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q2"


class D1Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q3"


class D1Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q4"


class D1Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D1-Q5"


class D2Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q1"


class D2Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q2"


class D2Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q3"
```

```
class D2Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q4"


class D2Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D2-Q5"


class D3Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q1"


class D3Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q2"


class D3Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q3"


class D3Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q4"


class D3Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D3-Q5"


class D4Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q1"


class D4Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q2"
```

```
class D4Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q3"


class D4Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q4"


class D4Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D4-Q5"


class D5Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q1"


class D5Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q2"


class D5Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q3"


class D5Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q4"


class D5Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D5-Q5"


class D6Q1_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q1"
```

```
class D6Q2_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q2"


class D6Q3_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q3"


class D6Q4_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q4"


class D6Q5_Executor_Contract(BaseExecutorWithContract):
    @classmethod
    def get_base_slot(cls) -> str:
        return "D6-Q5"


# Aliases expected by core orchestrator
D1Q1_Executor = D1Q1_Executor_Contract
D1Q2_Executor = D1Q2_Executor_Contract
D1Q3_Executor = D1Q3_Executor_Contract
D1Q4_Executor = D1Q4_Executor_Contract
D1Q5_Executor = D1Q5_Executor_Contract
D2Q1_Executor = D2Q1_Executor_Contract
D2Q2_Executor = D2Q2_Executor_Contract
D2Q3_Executor = D2Q3_Executor_Contract
D2Q4_Executor = D2Q4_Executor_Contract
D2Q5_Executor = D2Q5_Executor_Contract
D3Q1_Executor = D3Q1_Executor_Contract
D3Q2_Executor = D3Q2_Executor_Contract
D3Q3_Executor = D3Q3_Executor_Contract
D3Q4_Executor = D3Q4_Executor_Contract
D3Q5_Executor = D3Q5_Executor_Contract
D4Q1_Executor = D4Q1_Executor_Contract
D4Q2_Executor = D4Q2_Executor_Contract
D4Q3_Executor = D4Q3_Executor_Contract
D4Q4_Executor = D4Q4_Executor_Contract
D4Q5_Executor = D4Q5_Executor_Contract
D5Q1_Executor = D5Q1_Executor_Contract
D5Q2_Executor = D5Q2_Executor_Contract
D5Q3_Executor = D5Q3_Executor_Contract
D5Q4_Executor = D5Q4_Executor_Contract
D5Q5_Executor = D5Q5_Executor_Contract
D6Q1_Executor = D6Q1_Executor_Contract
D6Q2_Executor = D6Q2_Executor_Contract
```

```
D6Q3_Executor = D6Q3_Executor_Contract
D6Q4_Executor = D6Q4_Executor_Contract
D6Q5_Executor = D6Q5_Executor_Contract
```

src/farfan_pipeline/phases/Phase_two/generate_all_executor_configs.py

```
"""
Generate executor configuration files for all 30 D[1-6]Q[1-5] executors.

Creates individual JSON config files with runtime parameters (HOW)
for each executor, ensuring NO hardcoded calibration values (WHAT).
"""

import json
from pathlib import Path
from typing import Dict, List, Any

# Define all 30 executors with their metadata
EXECUTORS = [
    # D1: DIAGNOSTICS & INPUTS
      ("D1_Q1_QuantitativeBaselineExtractor", "D1", "Q1", ["structural", "statistical",
"semantic"], 15),
        ("D1_Q2_ProblemDimensioningAnalyzer", "D1", "Q2", ["bayesian", "statistical",
"normative"], 12),
          ("D1_Q3_BudgetAllocationTracer", "D1", "Q3", ["structural", "financial",
"normative"], 13),
      ("D1_Q4_InstitutionalCapacityIdentifier", "D1", "Q4", ["semantic", "structural"],
11),
        ("D1_Q5_ScopeJustificationValidator", "D1", "Q5", ["temporal", "consistency",
"structural"], 7),

    # D2: ACTIVITY DESIGN
    ("D2_Q1_StructuredPlanningValidator", "D2", "Q1", ["structural", "normative"], 7),
          ("D2_Q2_InterventionLogicInferencer", "D2", "Q2", ["causal", "bayesian",
"structural"], 11),
     ("D2_Q3_RootCauseLinkageAnalyzer", "D2", "Q3", ["causal", "bayesian", "semantic"],
9),
          ("D2_Q4_RiskManagementAnalyzer", "D2", "Q4", ["bayesian", "statistical",
"normative"], 10),
       ("D2_Q5_StrategicCoherenceEvaluator", "D2", "Q5", ["normative", "consistency",
"statistical"], 8),

    # D3: PRODUCTS & OUTPUTS
          ("D3_Q1_IndicatorQualityValidator", "D3", "Q1", ["normative", "semantic",
"structural"], 8),
       ("D3_Q2_TargetProportionalityAnalyzer", "D3", "Q2", ["structural", "financial",
"bayesian"], 21),
    ("D3_Q3_TraceabilityValidator", "D3", "Q3", ["structural", "semantic", "normative"],
22),
        ("D3_Q4_TechnicalFeasibilityEvaluator", "D3", "Q4", ["structural", "causal",
"statistical"], 26),
          ("D3_Q5_OutputOutcomeLinkageAnalyzer", "D3", "Q5", ["causal", "bayesian",
"semantic"], 25),

    # D4: RESULTS & OUTCOMES
           ("D4_Q1_OutcomeMetricsValidator", "D4", "Q1", ["semantic", "temporal",
"statistical"], 17),
    ("D4_Q2_CausalChainValidator", "D4", "Q2", ["causal", "bayesian", "structural"], 8),
```

```
    ("D4_Q3_AmbitionJustificationAnalyzer", "D4", "Q3", ["bayesian", "statistical"], 8),
    ("D4_Q4_ProblemSolvencyEvaluator", "D4", "Q4", ["normative", "consistency"], 7),
    ("D4_Q5_VerticalAlignmentValidator", "D4", "Q5", ["normative", "structural"], 6),

    # D5: IMPACTS
    ("D5_Q1_LongTermVisionAnalyzer", "D5", "Q1", ["causal", "temporal", "semantic"], 8),
       ("D5_Q2_CompositeMeasurementValidator", "D5", "Q2", ["bayesian", "statistical",
"normative"], 14),
    ("D5_Q3_IntangibleMeasurementAnalyzer", "D5", "Q3", ["semantic", "normative"], 7),
     ("D5_Q4_SystemicRiskEvaluator", "D5", "Q4", ["bayesian", "causal", "statistical"],
9),
         ("D5_Q5_RealismAndSideEffectsAnalyzer", "D5", "Q5", ["causal", "bayesian",
"normative"], 9),

    # D6: CAUSAL THEORY
      ("D6_Q1_ExplicitTheoryBuilder", "D6", "Q1", ["causal", "structural", "semantic"],
10),
        ("D6_Q2_LogicalProportionalityValidator", "D6", "Q2", ["normative", "causal",
"statistical"], 8),
           ("D6_Q3_ValidationTestingAnalyzer", "D6", "Q3", ["bayesian", "causal",
"statistical"], 9),
    ("D6_Q4_FeedbackLoopAnalyzer", "D6", "Q4", ["causal", "temporal", "structural"], 8),
       ("D6_Q5_ContextualAdaptabilityEvaluator", "D6", "Q5", ["semantic", "normative",
"contextual"], 8),
]


def generate_executor_config(
    executor_id: str,
    dimension: str,
    question: str,
    epistemic_mix: List[str],
    expected_methods: int
) -> Dict[str, Any]:
    """Generate configuration for a single executor."""

    # Base config template
    config = {
        "executor_id": executor_id,
        "dimension": dimension,
        "question": question,
                                                                "canonical_label":
f"DIM{dimension[1:].zfill(2)}_Q{question[1:].zfill(2)}_{executor_id.split('_',
2)[2].upper()}",
        "role": "SCORE_Q",
        "required_layers": [
            "@b",      # BASE: Code quality
            "@chain",  # CHAIN: Method wiring
            "@q",      # QUESTION: Question appropriateness
            "@d",      # DIMENSION: Dimension alignment
            "@p",      # POLICY: Policy area fit
            "@C",      # CONGRUENCE: Contract compliance
            "@u",      # UNIT: Document quality
            "@m",      # META: Governance maturity
```

```
        ],
        "runtime_parameters": {
            "timeout_s": 300,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": 512,
            "enable_profiling": True
        },
        "thresholds": {
            "min_quality_score": 0.5,
            "min_evidence_confidence": 0.6,
            "max_runtime_ms": 60000
        },
        "epistemic_mix": epistemic_mix,
        "contextual_params": {
            "expected_methods": expected_methods,
            "critical_methods": []
        },
        "calibration_settings": {
            "enabled": True,
            "capture_runtime_metrics": True,
            "capture_memory_metrics": True,
            "store_results": True
        }
    }

    return config


def main():
    """Generate all executor configuration files."""
    output_dir = Path(__file__).parent / "executor_configs"
    output_dir.mkdir(exist_ok=True)

    generated_count = 0

    for executor_id, dimension, question, epistemic_mix, expected_methods in EXECUTORS:
        config = generate_executor_config(
            executor_id, dimension, question, epistemic_mix, expected_methods
        )

        output_file = output_dir / f"{executor_id}.json"
        with open(output_file, 'w') as f:
            json.dump(config, f, indent=2)

        print(f"Generated: {output_file.name}")
        generated_count += 1

    print(f"\nTotal configs generated: {generated_count}")
    print(f"Expected: 30")
    print(f"Status: {'? COMPLETE' if generated_count == 30 else '? INCOMPLETE'}")
```

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

```
src/farfan_pipeline/phases/Phase_two/generate_all_executor_configs_complete.py

"""
Generate complete executor configuration files for all 30 D[1-6]Q[1-5] executors.

This script creates comprehensive JSON config files for each executor that separate:
- Calibration data (WHAT quality scores) ? loaded from intrinsic_calibration.json
- Runtime parameters (HOW execution) ? stored in executor_configs/{executor_id}.json

Ensures NO hardcoded calibration values in executor code.
"""

from __future__ import annotations

import json
from pathlib import Path
from typing import Dict, List, Any

EXECUTOR_DEFINITIONS = [
    {
        "executor_id": "D1_Q1_QuantitativeBaselineExtractor",
        "dimension": "D1",
        "question": "Q1",
        "canonical_label": "Extracción de Línea Base Cuantitativa",
        "methods_count": 15,
        "epistemic_mix": ["semantic", "statistical", "normative"],
    },
    {
        "executor_id": "D1_Q2_ProblemDimensioningAnalyzer",
        "dimension": "D1",
        "question": "Q2",
        "canonical_label": "Dimensionamiento del Problema",
        "methods_count": 12,
        "epistemic_mix": ["bayesian", "statistical", "normative"],
    },
    {
        "executor_id": "D1_Q3_BudgetAllocationTracer",
        "dimension": "D1",
        "question": "Q3",
        "canonical_label": "Trazabilidad de Asignación Presupuestal",
        "methods_count": 13,
        "epistemic_mix": ["structural", "financial", "normative"],
    },
    {
        "executor_id": "D1_Q4_InstitutionalCapacityIdentifier",
        "dimension": "D1",
        "question": "Q4",
        "canonical_label": "Identificación de Capacidad Institucional",
        "methods_count": 11,
        "epistemic_mix": ["semantic", "structural"],
    },
    {
        "executor_id": "D1_Q5_ScopeJustificationValidator",
        "dimension": "D1",
```

```
    "question": "Q5",
    "canonical_label": "Validación de Justificación de Alcance",
    "methods_count": 7,
    "epistemic_mix": ["temporal", "consistency", "normative"],
},
{
    "executor_id": "D2_Q1_StructuredPlanningValidator",
    "dimension": "D2",
    "question": "Q1",
    "canonical_label": "Validación de Planificación Estructurada",
    "methods_count": 7,
    "epistemic_mix": ["structural", "normative"],
},
{
    "executor_id": "D2_Q2_InterventionLogicInferencer",
    "dimension": "D2",
    "question": "Q2",
    "canonical_label": "Inferencia de Lógica de Intervención",
    "methods_count": 11,
    "epistemic_mix": ["causal", "bayesian", "structural"],
},
{
    "executor_id": "D2_Q3_RootCauseLinkageAnalyzer",
    "dimension": "D2",
    "question": "Q3",
    "canonical_label": "Análisis de Vinculación a Causas Raíz",
    "methods_count": 9,
    "epistemic_mix": ["causal", "structural", "semantic"],
},
{
    "executor_id": "D2_Q4_RiskManagementAnalyzer",
    "dimension": "D2",
    "question": "Q4",
    "canonical_label": "Análisis de Gestión de Riesgos",
    "methods_count": 10,
    "epistemic_mix": ["bayesian", "statistical", "normative"],
},
{
    "executor_id": "D2_Q5_StrategicCoherenceEvaluator",
    "dimension": "D2",
    "question": "Q5",
    "canonical_label": "Evaluación de Coherencia Estratégica",
    "methods_count": 8,
    "epistemic_mix": ["consistency", "normative", "structural"],
},
{
    "executor_id": "D3_Q1_IndicatorQualityValidator",
    "dimension": "D3",
    "question": "Q1",
    "canonical_label": "Validación de Calidad de Indicadores",
    "methods_count": 8,
    "epistemic_mix": ["normative", "structural", "semantic"],
},
{
```

```
    "executor_id": "D3_Q2_TargetProportionalityAnalyzer",
    "dimension": "D3",
    "question": "Q2",
    "canonical_label": "DIM03_Q02_PRODUCT_TARGET_PROPORTIONALITY",
    "methods_count": 21,
    "epistemic_mix": ["statistical", "financial", "normative"],
},
{
    "executor_id": "D3_Q3_TraceabilityValidator",
    "dimension": "D3",
    "question": "Q3",
    "canonical_label": "DIM03_Q03_TRACEABILITY_BUDGET_ORG",
    "methods_count": 22,
    "epistemic_mix": ["structural", "semantic", "normative"],
},
{
    "executor_id": "D3_Q4_TechnicalFeasibilityEvaluator",
    "dimension": "D3",
    "question": "Q4",
    "canonical_label": "DIM03_Q04_TECHNICAL_FEASIBILITY",
    "methods_count": 10,
    "epistemic_mix": ["financial", "normative", "statistical"],
},
{
    "executor_id": "D3_Q5_OutputOutcomeLinkageAnalyzer",
    "dimension": "D3",
    "question": "Q5",
    "canonical_label": "DIM03_Q05_OUTPUT_OUTCOME_LINKAGE",
    "methods_count": 9,
    "epistemic_mix": ["causal", "structural", "semantic"],
},
{
    "executor_id": "D4_Q1_OutcomeMetricsValidator",
    "dimension": "D4",
    "question": "Q1",
    "canonical_label": "DIM04_Q01_OUTCOME_INDICATOR_COMPLETENESS",
    "methods_count": 8,
    "epistemic_mix": ["normative", "statistical", "semantic"],
},
{
    "executor_id": "D4_Q2_CausalChainValidator",
    "dimension": "D4",
    "question": "Q2",
    "canonical_label": "Validación de Cadena Causal",
    "methods_count": 10,
    "epistemic_mix": ["causal", "structural", "consistency"],
},
{
    "executor_id": "D4_Q3_AmbitionJustificationAnalyzer",
    "dimension": "D4",
    "question": "Q3",
    "canonical_label": "Análisis de Justificación de Ambición",
    "methods_count": 9,
    "epistemic_mix": ["normative", "statistical", "semantic"],
```

```
},
{
    "executor_id": "D4_Q4_ProblemSolvencyEvaluator",
    "dimension": "D4",
    "question": "Q4",
    "canonical_label": "Evaluación de Solvencia del Problema",
    "methods_count": 11,
    "epistemic_mix": ["causal", "bayesian", "normative"],
},
{
    "executor_id": "D4_Q5_VerticalAlignmentValidator",
    "dimension": "D4",
    "question": "Q5",
    "canonical_label": "Validación de Alineación Vertical",
    "methods_count": 8,
    "epistemic_mix": ["structural", "consistency", "normative"],
},
{
    "executor_id": "D5_Q1_LongTermVisionAnalyzer",
    "dimension": "D5",
    "question": "Q1",
    "canonical_label": "Análisis de Visión de Largo Plazo",
    "methods_count": 8,
    "epistemic_mix": ["semantic", "normative", "temporal"],
},
{
    "executor_id": "D5_Q2_CompositeMeasurementValidator",
    "dimension": "D5",
    "question": "Q2",
    "canonical_label": "DIM05_Q02_COMPOSITE_PROXY_VALIDITY",
    "methods_count": 10,
    "epistemic_mix": ["statistical", "normative", "semantic"],
},
{
    "executor_id": "D5_Q3_IntangibleMeasurementAnalyzer",
    "dimension": "D5",
    "question": "Q3",
    "canonical_label": "Análisis de Medición Intangible",
    "methods_count": 9,
    "epistemic_mix": ["semantic", "normative", "bayesian"],
},
{
    "executor_id": "D5_Q4_SystemicRiskEvaluator",
    "dimension": "D5",
    "question": "Q4",
    "canonical_label": "Evaluación de Riesgo Sistémico",
    "methods_count": 12,
    "epistemic_mix": ["bayesian", "causal", "normative"],
},
{
    "executor_id": "D5_Q5_RealismAndSideEffectsAnalyzer",
    "dimension": "D5",
    "question": "Q5",
    "canonical_label": "Análisis de Realismo y Efectos Colaterales",
```

```
        "methods_count": 11,
        "epistemic_mix": ["causal", "bayesian", "normative"],
    },
    {
        "executor_id": "D6_Q1_ExplicitTheoryBuilder",
        "dimension": "D6",
        "question": "Q1",
        "canonical_label": "Constructor de Teoría Explícita",
        "methods_count": 15,
        "epistemic_mix": ["causal", "structural", "semantic"],
    },
    {
        "executor_id": "D6_Q2_LogicalProportionalityValidator",
        "dimension": "D6",
        "question": "Q2",
        "canonical_label": "Validación de Proporcionalidad Lógica",
        "methods_count": 10,
        "epistemic_mix": ["structural", "consistency", "normative"],
    },
    {
        "executor_id": "D6_Q3_ValidationTestingAnalyzer",
        "dimension": "D6",
        "question": "Q3",
        "canonical_label": "Análisis de Pruebas de Validación",
        "methods_count": 13,
        "epistemic_mix": ["causal", "bayesian", "statistical"],
    },
    {
        "executor_id": "D6_Q4_FeedbackLoopAnalyzer",
        "dimension": "D6",
        "question": "Q4",
        "canonical_label": "Análisis de Bucles de Retroalimentación",
        "methods_count": 9,
        "epistemic_mix": ["causal", "structural", "temporal"],
    },
    {
        "executor_id": "D6_Q5_ContextualAdaptabilityEvaluator",
        "dimension": "D6",
        "question": "Q5",
        "canonical_label": "Evaluación de Adaptabilidad Contextual",
        "methods_count": 10,
        "epistemic_mix": ["semantic", "normative", "causal"],
    },
]


def generate_executor_config(executor_def: Dict[str, Any]) -> Dict[str, Any]:
    """
    Generate complete executor config with runtime parameters only.

    NO calibration values (quality scores) are stored here.
    Calibration data is loaded from intrinsic_calibration.json.
    """
    methods_count = executor_def["methods_count"]
```

```
    timeout_base = 300
    if methods_count > 15:
        timeout_base = 600
    elif methods_count > 20:
        timeout_base = 900

    memory_limit_base = 512
             if   "causal"   in   executor_def["epistemic_mix"]   or   "bayesian"   in
executor_def["epistemic_mix"]:
        memory_limit_base = 1024

    config = {
        "executor_id": executor_def["executor_id"],
        "dimension": executor_def["dimension"],
        "question": executor_def["question"],
        "canonical_label": executor_def["canonical_label"],
        "role": "SCORE_Q",
        "required_layers": [
            "@b",
            "@chain",
            "@q",
            "@d",
            "@p",
            "@C",
            "@u",
            "@m"
        ],
        "runtime_parameters": {
            "timeout_s": timeout_base,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": memory_limit_base,
            "enable_profiling": True,
            "seed": 42
        },
        "thresholds": {
            "min_quality_score": 0.5,
            "min_evidence_confidence": 0.7,
            "max_runtime_ms": timeout_base * 1000
        },
        "epistemic_mix": executor_def["epistemic_mix"],
        "contextual_params": {
            "expected_methods": methods_count,
            "critical_methods": [],
            "dimension_label": f"DIM0{executor_def['dimension'][1]}",
            "question_label": executor_def["canonical_label"]
        }
    }

    return config
```

```
def main():
    """Generate all 30 executor config files."""
    output_dir = Path(__file__).parent / "executor_configs"
    output_dir.mkdir(exist_ok=True)

    print(f"Generating {len(EXECUTOR_DEFINITIONS)} executor config files...")

    for executor_def in EXECUTOR_DEFINITIONS:
        config = generate_executor_config(executor_def)
        output_file = output_dir / f"{executor_def['executor_id']}.json"

        with open(output_file, "w") as f:
            json.dump(config, f, indent=2, ensure_ascii=False)

        print(f"  ? {executor_def['executor_id']}.json")

    template_config = {
        "executor_id": "D{X}_Q{Y}_ExecutorName",
        "dimension": "D{X}",
        "question": "Q{Y}",
        "canonical_label": "Human-readable label",
        "role": "SCORE_Q",
        "required_layers": ["@b", "@chain", "@q", "@d", "@p", "@C", "@u", "@m"],
        "runtime_parameters": {
            "timeout_s": 300,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": 512,
            "enable_profiling": True,
            "seed": 42
        },
        "thresholds": {
            "min_quality_score": 0.5,
            "min_evidence_confidence": 0.7,
            "max_runtime_ms": 300000
        },
        "epistemic_mix": ["semantic", "statistical", "normative"],
        "contextual_params": {
            "expected_methods": 10,
            "critical_methods": [],
            "dimension_label": "DIM0X",
            "question_label": "Label"
        }
    }

    template_file = output_dir / "executor_config_template.json"
    with open(template_file, "w") as f:
        json.dump(template_config, f, indent=2, ensure_ascii=False)

    print(f"  ? executor_config_template.json")
    print(f"\n? Generated {len(EXECUTOR_DEFINITIONS)} executor configs + 1 template")
    print(f"? Output directory: {output_dir}")
    print("\n??  IMPORTANT: These files contain ONLY runtime parameters (HOW).")
```

```
    print("    Calibration data (WHAT quality scores) is loaded from:")
                                        print("                                                -
src/cross_cutting_infrastructure/capaz_calibration_parmetrization/calibration/COHORT_202
4_intrinsic_calibration.json")
    print("    - canonic_questionnaire_central/questionnaire_monolith.json")


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

```
src/farfan_pipeline/phases/Phase_two/generate_executor_configs.py

"""
Generate executor-specific configuration files for all 30+ D[1-6]Q[1-5] executors.

This script creates JSON configuration files with runtime parameters (HOW)
for each executor, ensuring NO hardcoded calibration values (WHAT).
"""

import json
from pathlib import Path
from typing import Dict, List, Any

# Executor metadata: (dimension, question, name, epistemic_mix, critical_methods)
EXECUTORS = [
    # Dimension 1: Diagnostics & Inputs
          ("D1",  "Q1",  "QuantitativeBaselineExtractor",  ["semantic",  "statistical",
"normative"], 15),
           ("D1",  "Q2",  "ProblemDimensioningAnalyzer",  ["bayesian",  "statistical",
"normative"], 12),
      ("D1", "Q3", "BudgetAllocationTracer", ["structural", "financial", "normative"],
13),
    ("D1", "Q4", "InstitutionalCapacityIdentifier", ["semantic", "structural"], 11),
           ("D1",  "Q5",  "ScopeJustificationValidator",  ["temporal",  "consistency",
"normative"], 7),

    # Dimension 2: Activity Design
    ("D2", "Q1", "StructuredPlanningValidator", ["structural", "normative"], 7),
      ("D2", "Q2", "InterventionLogicInferencer", ["causal", "bayesian", "structural"],
11),
    ("D2", "Q3", "RootCauseLinkageAnalyzer", ["causal", "structural", "semantic"], 9),
      ("D2", "Q4", "RiskManagementAnalyzer", ["bayesian", "statistical", "normative"],
10),
          ("D2",  "Q5",  "StrategicCoherenceEvaluator",  ["consistency",  "normative",
"structural"], 8),

    # Dimension 3: Products & Outputs
     ("D3", "Q1", "IndicatorQualityValidator", ["normative", "structural", "semantic"],
8),
         ("D3",  "Q2",  "TargetProportionalityAnalyzer",  ["statistical",  "financial",
"normative"], 21),
    ("D3", "Q3", "TraceabilityValidator", ["structural", "semantic", "normative"], 22),
           ("D3",  "Q4",  "TechnicalFeasibilityEvaluator",  ["financial",  "normative",
"statistical"], 10),
     ("D3", "Q5", "OutputOutcomeLinkageAnalyzer", ["causal", "structural", "semantic"],
9),

    # Dimension 4: Outcomes
      ("D4", "Q1", "OutcomeMetricsValidator", ["normative", "statistical", "semantic"],
8),
    ("D4", "Q2", "CausalChainValidator", ["causal", "structural", "consistency"], 10),
         ("D4",  "Q3",  "AmbitionJustificationAnalyzer",  ["normative",  "statistical",
"semantic"], 9),
    ("D4", "Q4", "ProblemSolvencyEvaluator", ["causal", "bayesian", "normative"], 11),
```

```
          ("D4",  "Q5",  "VerticalAlignmentValidator",  ["structural",  "consistency",
"normative"], 8),

    # Dimension 5: Impacts
    ("D5", "Q1", "LongTermVisionAnalyzer", ["semantic", "normative", "temporal"], 8),
          ("D5",  "Q2",  "CompositeMeasurementValidator",  ["statistical",  "normative",
"semantic"], 10),
    ("D5", "Q3", "IntangibleMeasurementAnalyzer", ["semantic", "normative", "bayesian"],
9),
    ("D5", "Q4", "SystemicRiskEvaluator", ["bayesian", "causal", "normative"], 12),
     ("D5", "Q5", "RealismAndSideEffectsAnalyzer", ["causal", "bayesian", "normative"],
11),

    # Dimension 6: Theory of Change
    ("D6", "Q1", "ExplicitTheoryBuilder", ["causal", "structural", "semantic"], 15),
        ("D6",  "Q2",  "LogicalProportionalityValidator",  ["structural",  "consistency",
"normative"], 10),
      ("D6",  "Q3",  "ValidationTestingAnalyzer",  ["causal",  "bayesian",  "statistical"],
13),
    ("D6", "Q4", "FeedbackLoopAnalyzer", ["causal", "structural", "temporal"], 9),
    ("D6", "Q5", "ContextualAdaptabilityEvaluator", ["semantic", "normative", "causal"],
10),
]


def create_executor_config(
    dimension: str,
    question: str,
    name: str,
    epistemic_mix: List[str],
    expected_methods: int
) -> Dict[str, Any]:
    """
    Create configuration dict for an executor.

    Args:
        dimension: Dimension ID (D1-D6)
        question: Question ID (Q1-Q5)
        name: Executor class name suffix
        epistemic_mix: List of epistemic tags
        expected_methods: Expected number of methods

    Returns:
        Configuration dictionary
    """
    executor_id = f"{dimension}_{question}_{name}"

    # Base runtime parameters (HOW - parametrization)
    config = {
        "executor_id": executor_id,
        "dimension": dimension,
        "question": question,
        "role": "SCORE_Q",
        "required_layers": [
```

```
            "@b",      # BASE: Code quality
            "@chain",  # CHAIN: Method wiring
            "@q",      # QUESTION: Question appropriateness
            "@d",      # DIMENSION: Dimension alignment
            "@p",      # POLICY: Policy area fit
            "@C",      # CONGRUENCE: Contract compliance
            "@u",      # UNIT: Document quality
            "@m",      # META: Governance maturity
        ],
        "runtime_parameters": {
            "timeout_s": 300,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": 512,
            "enable_profiling": True,
        },
        "thresholds": {
            "min_quality_score": 0.5,
            "min_evidence_confidence": 0.6,
            "max_runtime_ms": 60000,
        },
        "epistemic_mix": epistemic_mix,
        "contextual_params": {
            "expected_methods": expected_methods,
            "critical_methods": [],
        },
    }

    # Dimension-specific adjustments
    if dimension in ["D5", "D6"]:
        # Complex causal analysis: higher timeout, more tokens
        config["runtime_parameters"]["timeout_s"] = 450
        config["runtime_parameters"]["max_tokens"] = 8192
        config["runtime_parameters"]["memory_limit_mb"] = 1024

    if dimension in ["D1", "D2"]:
        # Input/baseline analysis: stricter thresholds
        config["thresholds"]["min_evidence_confidence"] = 0.7

    return config


def generate_all_configs(output_dir: Path) -> None:
    """
    Generate configuration files for all executors.

    Args:
        output_dir: Directory to write config files
    """
    output_dir.mkdir(parents=True, exist_ok=True)

    report = {
        "metadata": {
```

```
            "description": "Executor configuration generation report",
            "total_executors": len(EXECUTORS),
        },
        "executors": []
    }

    for dimension, question, name, epistemic_mix, expected_methods in EXECUTORS:
        config = create_executor_config(
            dimension, question, name, epistemic_mix, expected_methods
        )

        executor_id = config["executor_id"]
        output_file = output_dir / f"{executor_id}.json"

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

        report["executors"].append({
            "executor_id": executor_id,
            "config_file": str(output_file),
            "dimension": dimension,
            "question": question,
            "epistemic_mix": epistemic_mix,
        })

        print(f"? Generated config: {output_file}")

    # Write summary report
    report_file = output_dir / "generation_report.json"
    with open(report_file, "w") as f:
        json.dump(report, f, indent=2)

    print(f"\n? Generated {len(EXECUTORS)} executor configs")
    print(f"? Report: {report_file}")


if __name__ == "__main__":
    output_dir = Path(__file__).parent / "executor_configs"
    generate_all_configs(output_dir)
```

```
src/farfan_pipeline/phases/Phase_two/irrigation_synchronizer.py

"""Irrigation Synchronizer - Question?Chunk?Task?Plan Coordination.

This module implements the synchronization layer that maps questionnaire questions
to document chunks, generating an ExecutionPlan with 300 tasks (6 dimensions × 50
questions/dimension × 10 policy areas) for deterministic pipeline execution.

Architecture:
- IrrigationSynchronizer: Orchestrates chunk?question?task?plan flow
- ExecutionPlan: Immutable plan with deterministic plan_id and integrity_hash
- Task: Single unit of work (question + chunk + policy_area)
- Observability: Structured JSON logs with correlation_id tracking

Design Principles:
- Deterministic task generation (stable ordering, reproducible plan_id)
- Full observability (correlation_id propagates through all 10 phases)
- Prometheus metrics for synchronization health
- Blake3-based integrity hashing for plan verification
"""

from __future__ import annotations

import hashlib
import json
import logging
import statistics
import time
import uuid
from collections import Counter
from dataclasses import dataclass, field
from pathlib import Path
from typing import TYPE_CHECKING, Any, Protocol

if TYPE_CHECKING:
        from  cross_cutting_infrastructure.irrigation_using_signals.SISAS.signals  import
SignalRegistry

from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from orchestration.task_planner import ExecutableTask
from canonic_phases.Phase_two.phase6_validation import (
    validate_phase6_schema_compatibility,
)
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from farfan_pipeline.synchronization import ChunkMatrix

# Import executor-chunk synchronizer for JOIN table
try:
    from orchestration.executor_chunk_synchronizer import (
        ExecutorChunkBinding,
        build_join_table,
        generate_verification_manifest,
```

```
        save_verification_manifest,
        ExecutorChunkSynchronizationError,
    )
    SYNCHRONIZER_AVAILABLE = True
except ImportError as e:
    SYNCHRONIZER_AVAILABLE = False
    _import_error = e

    # Provide clear error messages when attempting to use unavailable features
    class ExecutorChunkBinding:  # type: ignore
        def __init__(self, *args: Any, **kwargs: Any) -> None:
            raise ImportError(
                "orchestration.executor_chunk_synchronizer is not available. "
                "Please ensure the dependency is installed and importable."
            ) from _import_error

    def build_join_table(*args: Any, **kwargs: Any) -> Any:
        raise ImportError(
            "orchestration.executor_chunk_synchronizer is not available. "
            "Please ensure the dependency is installed and importable."
        ) from _import_error

    def generate_verification_manifest(*args: Any, **kwargs: Any) -> Any:
        raise ImportError(
            "orchestration.executor_chunk_synchronizer is not available. "
            "Please ensure the dependency is installed and importable."
        ) from _import_error

    def save_verification_manifest(*args: Any, **kwargs: Any) -> Any:
        raise ImportError(
            "orchestration.executor_chunk_synchronizer is not available. "
            "Please ensure the dependency is installed and importable."
        ) from _import_error

    class ExecutorChunkSynchronizationError(Exception):  # type: ignore
        pass

try:
    from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signals import (
        SignalRegistry as _SignalRegistry,
    )
except ImportError:
    _SignalRegistry = None  # type: ignore

try:
    import blake3

    BLAKE3_AVAILABLE = True
except ImportError:
    BLAKE3_AVAILABLE = False

try:
    from prometheus_client import Counter, Histogram
```

```
    PROMETHEUS_AVAILABLE = True
except ImportError:
    PROMETHEUS_AVAILABLE = False

logger = logging.getLogger(__name__)

SHA256_HEX_DIGEST_LENGTH = 64

SKEW_THRESHOLD_CV = 0.3

class SignalRegistry(Protocol):
    """Protocol for signal registry implementations.

    Defines the interface that signal registries must implement for
    use with IrrigationSynchronizer signal resolution.
    """

    def get_signals_for_chunk(
        self, chunk: ChunkData, requirements: list[str]
    ) -> list[Any]:
        """Get signals for a chunk matching the given requirements.

        Args:
            chunk: Target chunk to get signals for
            requirements: List of required signal types

        Returns:
            List of signals, each with signal_id, signal_type, and content fields
        """
        ...


if PROMETHEUS_AVAILABLE:
    synchronization_duration = Histogram(
        "synchronization_duration_seconds",
        "Time spent building execution plan",
        buckets=[0.1, 0.5, 1.0, 2.0, 5.0, 10.0],
    )
    tasks_constructed = Counter(
        "synchronization_tasks_constructed_total",
        "Total number of tasks constructed",
        ["dimension", "policy_area"],
    )
    synchronization_failures = Counter(
        "synchronization_failures_total",
        "Total synchronization failures",
        ["error_type"],
    )
    synchronization_chunk_matches = Counter(
        "synchronization_chunk_matches_total",
        "Total chunk routing matches during synchronization",
        ["dimension", "policy_area", "status"],
    )
else:
```

```
    class DummyMetric:
        def time(self):
            class DummyContextManager:
                def __call__(self, func):
                    def wrapper(*args, **kwargs):
                        return func(*args, **kwargs)

                    return wrapper

                def __enter__(self):
                    return self

                def __exit__(self, *args):
                    pass

            return DummyContextManager()

        def labels(self, **kwargs):
            return self

        def inc(self, *args, **kwargs) -> None:
            pass

    synchronization_duration = DummyMetric()
    tasks_constructed = DummyMetric()
    synchronization_failures = DummyMetric()
    synchronization_chunk_matches = DummyMetric()

SHA256_HEX_DIGEST_LENGTH = 64


@dataclass(frozen=True)
class ChunkRoutingResult:
    """Result of Phase 3 chunk routing verification.

    Contains validated chunk reference and extracted metadata for task construction.
    """

    target_chunk: ChunkData
    chunk_id: str
    policy_area_id: str
    dimension_id: str
    text_content: str
    expected_elements: list[dict[str, Any]]
    document_position: tuple[int, int] | None


@dataclass(frozen=True)
class Task:
    """Single unit of work in the execution plan.

    Represents the mapping of one question to one chunk in a specific policy area.
    """
```

```
    task_id: str
    dimension: str
    question_id: str
    policy_area: str
    chunk_id: str
    chunk_index: int
    question_text: str


@dataclass
class ExecutionPlan:
    """Immutable execution plan with deterministic identifiers.

    Contains all tasks to be executed, with cryptographic integrity verification.
    """

    plan_id: str
    tasks: tuple[Task, ...]
    chunk_count: int
    question_count: int
    integrity_hash: str
    created_at: str
    correlation_id: str
    metadata: dict[str, Any] = field(default_factory=dict)

    def to_dict(self) -> dict[str, Any]:
        """Convert plan to dictionary for serialization."""
        return {
            "plan_id": self.plan_id,
            "tasks": [
                {
                    "task_id": t.task_id,
                    "dimension": t.dimension,
                    "question_id": t.question_id,
                    "policy_area": t.policy_area,
                    "chunk_id": t.chunk_id,
                    "chunk_index": t.chunk_index,
                    "question_text": t.question_text,
                }
                for t in self.tasks
            ],
            "chunk_count": self.chunk_count,
            "question_count": self.question_count,
            "integrity_hash": self.integrity_hash,
            "created_at": self.created_at,
            "correlation_id": self.correlation_id,
            "metadata": self.metadata,
        }

    @classmethod
    def from_dict(cls, data: dict[str, Any]) -> ExecutionPlan:
        """Reconstruct ExecutionPlan from dictionary.
```

```
        Args:
            data: Dictionary representation of ExecutionPlan

        Returns:
            ExecutionPlan instance reconstructed from dictionary
        """
        tasks = tuple(
            Task(
                task_id=t["task_id"],
                dimension=t["dimension"],
                question_id=t["question_id"],
                policy_area=t["policy_area"],
                chunk_id=t["chunk_id"],
                chunk_index=t["chunk_index"],
                question_text=t["question_text"],
            )
            for t in data["tasks"]
        )

        return cls(
            plan_id=data["plan_id"],
            tasks=tasks,
            chunk_count=data["chunk_count"],
            question_count=data["question_count"],
            integrity_hash=data["integrity_hash"],
            created_at=data["created_at"],
            correlation_id=data["correlation_id"],
        )


class IrrigationSynchronizer:
    """Synchronizes questionnaire questions with document chunks.

    Generates deterministic execution plans mapping questions to chunks across
    all policy areas, with full observability and integrity verification.
    """

    def __init__(
        self,
        questionnaire: dict[str, Any],
        preprocessed_document: PreprocessedDocument | None = None,
        document_chunks: list[dict[str, Any]] | None = None,
        signal_registry: SignalRegistry | None = None,
        contracts: list[dict[str, Any]] | None = None,
        enable_join_table: bool = False,
    ) -> None:
        """Initialize synchronizer with questionnaire and chunks.

        Args:
            questionnaire: Loaded questionnaire_monolith.json data
            preprocessed_document: PreprocessedDocument containing validated chunks
            document_chunks: Legacy list of document chunks (deprecated)
              signal_registry: SignalRegistry for Phase 5 signal resolution (initialized
if None)
```

```
                contracts: Optional list of executor contracts for JOIN table
(Q001-Q300.v3.json)
            enable_join_table: Enable canonical JOIN table architecture (default: False)

        Raises:
            ValueError: If chunk matrix validation fails or no chunks provided
        """
        self.questionnaire = questionnaire
        self.correlation_id = str(uuid.uuid4())
        self.question_count = self._count_questions()
        self.chunk_matrix: ChunkMatrix | None = None
        self.document_chunks: list[dict[str, Any]] | None = None
        self.executor_contracts = contracts
        self.enable_join_table = enable_join_table and SYNCHRONIZER_AVAILABLE
        self.join_table: list[ExecutorChunkBinding] | None = None

        if signal_registry is None and _SignalRegistry is not None:
            self.signal_registry: SignalRegistry | None = _SignalRegistry()
        else:
            self.signal_registry = signal_registry

        if preprocessed_document is not None:
            try:
                self.chunk_matrix = ChunkMatrix(preprocessed_document)
                self.chunk_count = ChunkMatrix.EXPECTED_CHUNK_COUNT

                logger.info(
                    json.dumps(
                        {
                            "event": "irrigation_synchronizer_init",
                            "correlation_id": self.correlation_id,
                            "question_count": self.question_count,
                            "chunk_count": self.chunk_count,
                            "chunk_matrix_validated": True,
                            "mode": "preprocessed_document",
                            "timestamp": time.time(),
                        }
                    )
                )
            except ValueError as e:
                synchronization_failures.labels(
                    error_type="chunk_matrix_validation"
                ).inc()
                logger.error(
                    json.dumps(
                        {
                            "event": "irrigation_synchronizer_init_failed",
                            "correlation_id": self.correlation_id,
                            "error": str(e),
                            "error_type": "chunk_matrix_validation",
                            "timestamp": time.time(),
                        }
                    )
                )
```

```
                raise ValueError(
                     f"Chunk matrix validation failed during synchronizer initialization:
{e}"
                ) from e
        elif document_chunks is not None:
            self.document_chunks = document_chunks
            self.chunk_count = len(document_chunks)

            logger.info(
                json.dumps(
                    {
                        "event": "irrigation_synchronizer_init",
                        "correlation_id": self.correlation_id,
                        "question_count": self.question_count,
                        "chunk_count": self.chunk_count,
                        "mode": "legacy_document_chunks",
                        "timestamp": time.time(),
                    }
                )
            )
        else:
            raise ValueError(
                "Either preprocessed_document or document_chunks must be provided"
            )

    def _count_questions(self) -> int:
        """Count total questions across all dimensions."""
        blocks = self.questionnaire.get("blocks", {})

        micro_questions = blocks.get("micro_questions")
        if isinstance(micro_questions, list):
            return len(micro_questions)

        count = 0
        for dimension_key in ["D1", "D2", "D3", "D4", "D5", "D6"]:
            for i in range(1, 51):
                question_key = f"D{dimension_key[1]}_Q{i:02d}"
                if question_key in blocks:
                    count += 1

        return count

    def validate_chunk_routing(self, question: dict[str, Any]) -> ChunkRoutingResult:
        """Phase 3: Validate chunk routing and extract metadata.

        Verifies that a chunk exists in the matrix for the question's routing keys,
        validates chunk consistency, and extracts metadata for task construction.

        Args:
            question: Question dict with routing keys (policy_area_id, dimension_id)

        Returns:
            ChunkRoutingResult with validated chunk and extracted metadata
```

```
        Raises:
            ValueError: If chunk not found or validation fails
        """
        question_id = question.get("question_id", "UNKNOWN")
        policy_area_id = question.get("policy_area_id")
        dimension_id = question.get("dimension_id")

        if not policy_area_id:
            raise ValueError(
                f"Question {question_id} missing required field: policy_area_id"
            )

        if not dimension_id:
            raise ValueError(
                f"Question {question_id} missing required field: dimension_id"
            )

        try:
            target_chunk = self.chunk_matrix.get_chunk(policy_area_id, dimension_id)

            chunk_id = target_chunk.chunk_id or f"{policy_area_id}-{dimension_id}"

            if not target_chunk.text or not target_chunk.text.strip():
                raise ValueError(
                                f"Chunk {chunk_id} has empty text content for question
{question_id}"
                )

            if (
                target_chunk.policy_area_id
                and target_chunk.policy_area_id != policy_area_id
            ):
                raise ValueError(
                    f"Chunk routing key mismatch for {question_id}: "
                                f"question policy_area={policy_area_id} but chunk has
{target_chunk.policy_area_id}"
                )

            if target_chunk.dimension_id and target_chunk.dimension_id != dimension_id:
                raise ValueError(
                    f"Chunk routing key mismatch for {question_id}: "
                                    f"question dimension={dimension_id} but chunk has
{target_chunk.dimension_id}"
                )

            expected_elements = question.get("expected_elements", [])

            document_position = None
            if target_chunk.start_pos is not None and target_chunk.end_pos is not None:
                document_position = (target_chunk.start_pos, target_chunk.end_pos)

            synchronization_chunk_matches.labels(
                dimension=dimension_id, policy_area=policy_area_id, status="success"
            ).inc()
```

```
        logger.debug(
            json.dumps(
                {
                    "event": "chunk_routing_success",
                    "question_id": question_id,
                    "chunk_id": chunk_id,
                    "policy_area_id": policy_area_id,
                    "dimension_id": dimension_id,
                    "text_length": len(target_chunk.text),
                    "has_expected_elements": len(expected_elements) > 0,
                    "has_document_position": document_position is not None,
                    "correlation_id": self.correlation_id,
                }
            )
        )

        return ChunkRoutingResult(
            target_chunk=target_chunk,
            chunk_id=chunk_id,
            policy_area_id=policy_area_id,
            dimension_id=dimension_id,
            text_content=target_chunk.text,
            expected_elements=expected_elements,
            document_position=document_position,
        )

    except KeyError as e:
        synchronization_chunk_matches.labels(
            dimension=dimension_id, policy_area=policy_area_id, status="failure"
        ).inc()

        error_msg = (
            f"Synchronization Failure for MQC {question_id}: "
            f"PA={policy_area_id}, DIM={dimension_id}. "
            f"No corresponding chunk found in matrix."
        )

        logger.error(
            json.dumps(
                {
                    "event": "chunk_routing_failure",
                    "question_id": question_id,
                    "policy_area_id": policy_area_id,
                    "dimension_id": dimension_id,
                    "error": error_msg,
                    "correlation_id": self.correlation_id,
                }
            )
        )

        raise ValueError(error_msg) from e

def _extract_questions(self) -> list[dict[str, Any]]:
```

```
        """Extract all questions from questionnaire in deterministic order."""
        questions = []

        # Ensure questionnaire is a dict (handle CanonicalQuestionnaire object)
        if not isinstance(self.questionnaire, dict):
                                        if   hasattr(self.questionnaire,   "data")   and
isinstance(self.questionnaire.data, dict):
                self.questionnaire = self.questionnaire.data
            elif hasattr(self.questionnaire, "to_dict"):
                self.questionnaire = self.questionnaire.to_dict()

        blocks = self.questionnaire.get("blocks", {})

                #  Fallback:  Check  if  micro_questions  is  at  root  or  directly  in
self.questionnaire
        micro_questions = blocks.get("micro_questions")
        if not micro_questions:
             micro_questions = self.questionnaire.get("micro_questions")
             if not micro_questions and isinstance(self.questionnaire, list):
                 # Handle case where questionnaire IS the list of questions
                 micro_questions = self.questionnaire

        if not blocks and not micro_questions:
            logger.warning("No 'blocks' found in questionnaire")

        if isinstance(micro_questions, list) and micro_questions:
                 logger.info(f"Found {len(micro_questions)} micro_questions in canonical
format")
            for raw in micro_questions:
                if not isinstance(raw, dict):
                    raise TypeError(
                        "Invalid micro_question type in questionnaire: "
                        f"expected dict but got {type(raw).__name__}"
                    )

                policy_area_id = raw.get("policy_area_id")
                dimension_id = raw.get("dimension_id")
                question_global = raw.get("question_global")

                if not isinstance(policy_area_id, str) or not policy_area_id:
                    raise ValueError(
                        "micro_question missing policy_area_id or invalid type: "
                        f"{policy_area_id!r}"
                    )
                if not isinstance(dimension_id, str) or not dimension_id:
                    raise ValueError(
                        "micro_question missing dimension_id or invalid type: "
                        f"{dimension_id!r}"
                    )
                if not isinstance(question_global, int):
                    raise ValueError(
                        "micro_question missing question_global or invalid type: "
                        f"{question_global!r}"
                    )
```

```
            patterns_raw = raw.get("patterns", [])
            patterns: list[dict[str, Any]] = []
            if isinstance(patterns_raw, list):
                for pattern in patterns_raw:
                    if not isinstance(pattern, dict):
                        raise TypeError(
                            "Invalid pattern type in micro_question: "
                            f"expected dict but got {type(pattern).__name__}"
                        )
                    enriched = dict(pattern)
                    enriched.setdefault("policy_area_id", policy_area_id)
                    patterns.append(enriched)

            questions.append(
                {
                    "question_id": raw.get("question_id"),
                    "question_global": question_global,
                    "question_text": raw.get("text", ""),
                    "policy_area_id": policy_area_id,
                    "dimension_id": dimension_id,
                    "base_slot": raw.get("base_slot", ""),
                    "cluster_id": raw.get("cluster_id", ""),
                    "patterns": patterns,
                    "expected_elements": raw.get("expected_elements", []),
                    "signal_requirements": raw.get("signal_requirements", []),
                    "validations": raw.get("validations", {}),
                }
            )

    questions.sort(key=lambda q: (q["policy_area_id"], q["question_global"]))
    return questions

for dimension in range(1, 7):
    dim_key = f"D{dimension}"
    dimension_id = f"DIM{dimension:02d}"

    for q_num in range(1, 51):
        question_key = f"{dim_key}_Q{q_num:02d}"

        if question_key in blocks:
            block = blocks[question_key]
            questions.append(
                {
                    "dimension": dim_key,
                    "question_id": question_key,
                    "question_num": q_num,
                    "question_global": block.get("question_global", 0),
                    "question_text": block.get("question", ""),
                    "policy_area_id": block.get("policy_area_id"),
                    "dimension_id": dimension_id,
                    "patterns": block.get("patterns", []),
                    "expected_elements": block.get("expected_elements", []),
                    "signal_requirements": block.get("signal_requirements", {}),
```

```
                    }
                )

        questions.sort(key=lambda q: (q["dimension_id"], q["question_id"]))

        return questions

    def _filter_patterns(
        self,
        patterns: list[dict[str, Any]] | tuple[dict[str, Any], ...],
        policy_area_id: str,
    ) -> tuple[dict[str, Any], ...]:
        """Filter patterns by policy_area_id using strict equality.

             Filters patterns to include only those where pattern.policy_area_id ==
policy_area_id
        (strict equality). Patterns lacking a policy_area_id attribute are excluded.

        Args:
                 patterns: Iterable of pattern objects (typically dicts with optional
policy_area_id)
            policy_area_id: Policy area ID string (e.g., "PA01") to filter by

        Returns:
                 Immutable tuple of filtered pattern dicts. Returns empty tuple if no
patterns match.

        Filtering Rules:
            - Strict equality: pattern.policy_area_id == policy_area_id
            - Exclude patterns without policy_area_id attribute
            - Result is immutable (tuple)
        """
        included = []
        excluded = []
        included_ids = []
        excluded_ids = []

        for pattern in patterns:
            pattern_id = (
                pattern.get("id", "UNKNOWN") if isinstance(pattern, dict) else "UNKNOWN"
            )

            if isinstance(pattern, dict) and "policy_area_id" in pattern:
                if pattern["policy_area_id"] == policy_area_id:
                    included.append(pattern)
                    included_ids.append(pattern_id)
                else:
                    excluded.append(pattern)
                    excluded_ids.append(pattern_id)
            else:
                excluded.append(pattern)
                excluded_ids.append(pattern_id)

        total_count = len(included) + len(excluded)
```

```
    logger.info(
        json.dumps(
            {
                "event": "IrrigationSynchronizer._filter_patterns",
                "total": total_count,
                "included": len(included),
                "excluded": len(excluded),
                "included_ids": included_ids,
                "excluded_ids": excluded_ids,
                "policy_area_id": policy_area_id,
                "correlation_id": self.correlation_id,
            }
        )
    )

    return tuple(included)

def _find_contract_for_question(
    self,
    question: dict[str, Any],
) -> dict[str, Any] | None:
    """Find executor contract for a given question.

    Args:
        question: Question dict with question_id

    Returns:
        Contract dict or None if not found
    """
    if not self.executor_contracts:
        return None

    question_id = question.get("question_id")
    if not question_id:
        return None

    # Try direct lookup by question_id (e.g., "D1_Q01" -> "Q001")
    # Extract global question number if available
    question_global = question.get("question_global")
    if question_global:
        contract_id = f"Q{question_global:03d}"
        for contract in self.executor_contracts:
            if contract.get("identity", {}).get("question_id") == contract_id:
                return contract

    # Fallback: match by policy_area_id and dimension_id
    policy_area_id = question.get("policy_area_id")
    dimension_id = question.get("dimension_id")
    if policy_area_id and dimension_id:
        for contract in self.executor_contracts:
            identity = contract.get("identity", {})
            if (identity.get("policy_area_id") == policy_area_id and
                identity.get("dimension_id") == dimension_id):
```

```
                    # Multiple contracts may match, return first
                    # In production, should have unique mapping
                    return contract

        return None

    def _filter_patterns_from_contract(
        self,
        contract: dict[str, Any],
        document_context: dict[str, Any] | None = None,
    ) -> tuple[dict[str, Any], ...]:
        """Filter patterns from contract using document context.

                        Contract-driven   pattern   irrigation:   uses   patterns   from
contract.question_context.patterns
        instead of generic questionnaire patterns. Provides higher precision (~85-90% vs
~60%).

        Args:
            contract: Executor contract (Q{nnn}.v3.json) with question_context.patterns
            document_context: Optional document context for advanced filtering

        Returns:
            Immutable tuple of filtered pattern dicts from contract
        """
        question_context = contract.get("question_context", {})
        patterns = question_context.get("patterns", [])

        if not document_context:
            # No context filtering, return all contract patterns
            logger.debug(
                json.dumps(
                    {
                                                                                "event":
"IrrigationSynchronizer._filter_patterns_from_contract",
                            "contract_id": contract.get("identity", {}).get("question_id",
"UNKNOWN"),
                        "total_patterns": len(patterns),
                        "filtering_mode": "no_context",
                        "correlation_id": self.correlation_id,
                    }
                )
            )
            return tuple(patterns)

        # Future: implement advanced context-based filtering
        # For now, return all contract patterns
        logger.info(
            json.dumps(
                {
                    "event": "IrrigationSynchronizer._filter_patterns_from_contract",
                          "contract_id": contract.get("identity", {}).get("question_id",
"UNKNOWN"),
                    "total_patterns": len(patterns),
```

```
                "filtering_mode": "contract_driven",
                "correlation_id": self.correlation_id,
            }
        )
    )
    return tuple(patterns)

def _build_join_table_if_enabled(
    self,
    chunks: list[Any],
) -> list[ExecutorChunkBinding] | None:
    """Build JOIN table if enabled and contracts available.

    Args:
        chunks: List of chunks from preprocessed_document

    Returns:
        List of ExecutorChunkBinding objects or None if not enabled

    Raises:
        ExecutorChunkSynchronizationError: If JOIN table construction fails
    """
    if not self.enable_join_table or not SYNCHRONIZER_AVAILABLE:
        return None

    if not self.executor_contracts:
        logger.warning(
            json.dumps(
                {
                    "event": "join_table_disabled",
                    "reason": "no_contracts_provided",
                    "correlation_id": self.correlation_id,
                }
            )
        )
        return None

    logger.info(
        json.dumps(
            {
                "event": "join_table_build_start",
                "contracts_count": len(self.executor_contracts),
                "chunks_count": len(chunks),
                "correlation_id": self.correlation_id,
                "timestamp": time.time(),
            }
        )
    )

    try:
        bindings = build_join_table(self.executor_contracts, chunks)

        logger.info(
            json.dumps(
```

```
                    {
                        "event": "join_table_build_success",
                        "bindings_count": len(bindings),
                        "correlation_id": self.correlation_id,
                        "timestamp": time.time(),
                    }
                )
            )

            return bindings

        except ExecutorChunkSynchronizationError as e:
            logger.error(
                json.dumps(
                    {
                        "event": "join_table_build_failed",
                        "error": str(e),
                        "correlation_id": self.correlation_id,
                        "timestamp": time.time(),
                    }
                )
            )
            raise

    def _construct_task(
        self,
        question: dict[str, Any],
        routing_result: ChunkRoutingResult,
        applicable_patterns: tuple[dict[str, Any], ...],
        resolved_signals: tuple[Any, ...],
        generated_task_ids: set[str],
    ) -> ExecutableTask:
        """Construct ExecutableTask from question and routing result.

            Extracts all fields from validated inputs, converts tuples to lists for
patterns,
             builds signals dict keyed by signal_type, generates creation_timestamp,
populates
          metadata with all required keys, validates all mandatory fields are non-None,
and
        catches TypeError from dataclass validation to re-raise as ValueError.

        Args:
            question: Question dict from questionnaire
            routing_result: Validated routing result from Phase 3
               applicable_patterns: Filtered tuple of patterns applicable to the routed
policy area
            resolved_signals: Resolved signals tuple from Phase 5
            generated_task_ids: Set of task IDs generated in current synchronization run

        Returns:
            ExecutableTask ready for execution

        Raises:
```

```
                ValueError: If duplicate task_id is detected or required fields are
missing/invalid
        """
        # Phase 7.1: Validate and extract question_global
        question_global = question.get("question_global")
        if question_global is None:
            raise ValueError("question_global field is required but missing")
        if not isinstance(question_global, int):
            raise ValueError(
                                        f"question_global must be an integer, got
{type(question_global).__name__}"
            )

        # Phase 7.1: Construct task_id from validated question_global
        task_id = f"MQC-{question_global:03d}_{routing_result.policy_area_id}"

        if task_id in generated_task_ids:
            raise ValueError(f"Duplicate task_id detected: {task_id}")

        generated_task_ids.add(task_id)

        # Field extraction in declaration order for validation priority
        # Extract question_id with bracket notation and KeyError conversion
        try:
            question_id = question["question_id"]
        except KeyError as e:
            raise ValueError("question_id field is required but missing") from e

        # Assign question_global (already validated above)
         # Extract routing fields via attribute access (guaranteed by ChunkRoutingResult
schema)
        policy_area_id = routing_result.policy_area_id
        dimension_id = routing_result.dimension_id
        chunk_id = routing_result.chunk_id

        expected_elements_list = list(routing_result.expected_elements)
        document_position = routing_result.document_position

        patterns_list = list(applicable_patterns)

        signals_dict: dict[str, Any] = {}
        for signal in resolved_signals:
            if isinstance(signal, dict) and "signal_type" in signal:
                signals_dict[signal["signal_type"]] = signal
            elif hasattr(signal, "signal_type"):
                signals_dict[signal.signal_type] = signal

        from datetime import datetime, timezone

        creation_timestamp = datetime.now(timezone.utc).isoformat()

        metadata = {
            "document_position": document_position,
            "synchronizer_version": "1.0.0",
```

```
            "correlation_id": self.correlation_id,
            "original_pattern_count": len(applicable_patterns),
            "original_signal_count": len(resolved_signals),
        }

        if task_id is None or not task_id:
            raise ValueError("Task construction failure: task_id is None or empty")
        if question_id is None or not question_id:
            raise ValueError("Task construction failure: question_id is None or empty")
        if question_global is None:
            raise ValueError("Task construction failure: question_global is None")
        if policy_area_id is None or not policy_area_id:
            raise ValueError(
                "Task construction failure: policy_area_id is None or empty"
            )
        if dimension_id is None or not dimension_id:
            raise ValueError("Task construction failure: dimension_id is None or empty")
        if chunk_id is None or not chunk_id:
            raise ValueError("Task construction failure: chunk_id is None or empty")
        if creation_timestamp is None or not creation_timestamp:
            raise ValueError(
                "Task construction failure: creation_timestamp is None or empty"
            )

        try:
            task = ExecutableTask(
                task_id=task_id,
                question_id=question_id,
                question_global=question_global,
                policy_area_id=policy_area_id,
                dimension_id=dimension_id,
                chunk_id=chunk_id,
                patterns=patterns_list,
                signals=signals_dict,
                creation_timestamp=creation_timestamp,
                expected_elements=expected_elements_list,
                metadata=metadata,
            )
        except TypeError as e:
            raise ValueError(
                  f"Task construction failed for {task_id}: dataclass validation error -
{e}"
            ) from e

        logger.debug(
            f"Constructed task: task_id={task_id}, question_id={question_id}, "
            f"chunk_id={chunk_id}, pattern_count={len(patterns_list)}, "
            f"signal_count={len(signals_dict)}"
        )

        return task

    def _assemble_execution_plan(
        self,
```

```
    executable_tasks: list[ExecutableTask],
    questions: list[dict[str, Any]],
    correlation_id: str,  # noqa: ARG002
) -> tuple[list[ExecutableTask], str]:
    """Phase 8: Assemble execution plan with validation and deterministic ordering.

    Performs four-phase assembly process:
    - Phase 8.1: Pre-assembly validation (duplicate detection, count validation)
    - Phase 8.2: Deterministic task ordering (lexicographic by task_id)
    - Phase 8.3: Plan identifier computation (SHA256 of deterministic JSON)
    - Phase 8.4: Plan identifier validation (format and length checks)

    Validates that task count matches question count and that no duplicate
    task identifiers exist. Then sorts tasks lexicographically by task_id to ensure
    deterministic plan identifier generation across runs. Computes plan_id by
    encoding deterministic JSON serialization (sort_keys=True, compact separators)
    to UTF-8 bytes, computing SHA256 hash, and validating result matches expected
    64-character lowercase hexadecimal format.

    Args:
        executable_tasks: List of constructed ExecutableTask objects
        questions: List of question dictionaries
        correlation_id: Correlation ID for tracing

    Returns:
        Tuple of (sorted list of ExecutableTask objects, plan_id string)

    Raises:
        ValueError: If task count doesn't match question count, duplicates exist,
                    or plan_id validation fails
        RuntimeError: When sorting operation corrupts task list length
    """
    from collections import Counter

    question_count = len(questions)
    task_count = len(executable_tasks)

    if task_count != question_count:
        raise ValueError(
            f"Execution plan assembly failure: expected {question_count} tasks "
            f"but constructed {task_count}; task construction loop corrupted"
        )

    task_ids = [t.task_id for t in executable_tasks]
    unique_count = len(set(task_ids))

    if unique_count != len(task_ids):
        counter = Counter(task_ids)
        duplicates = [task_id for task_id, count in counter.items() if count > 1]
        duplicate_count = len(task_ids) - unique_count

        raise ValueError(
            f"Execution plan assembly failure: found {duplicate_count} duplicate "
            f"task identifiers; duplicates are {sorted(duplicates)}"
```

```
            )

        sorted_tasks = sorted(executable_tasks, key=lambda t: t.task_id)

        if len(sorted_tasks) != len(executable_tasks):
            raise RuntimeError(
                                f"Task  ordering  corruption  detected:  sorted  task  count
{len(sorted_tasks)} "
                f"does not match input task count {len(executable_tasks)}"
            )

        task_serialization = [
            {
                "task_id": t.task_id,
                "question_id": t.question_id,
                "question_global": t.question_global,
                "policy_area_id": t.policy_area_id,
                "dimension_id": t.dimension_id,
                "chunk_id": t.chunk_id,
            }
            for t in sorted_tasks
        ]

        json_bytes = json.dumps(
            task_serialization, sort_keys=True, separators=(",", ":")
        ).encode("utf-8")

        plan_id = hashlib.sha256(json_bytes).hexdigest()

        if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
            raise ValueError(
                                  f"Plan  identifier  validation  failure:  expected  length
{SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
                "SHA256 implementation may be compromised or monkey-patched"
            )

        if not all(c in "0123456789abcdef" for c in plan_id):
            raise ValueError(
                 "Plan identifier validation failure: expected lowercase hexadecimal but
got "
                 "characters outside '0123456789abcdef' set; SHA256 implementation may be
"
                "compromised or monkey-patched"
            )

        return sorted_tasks, plan_id

    def _compute_integrity_hash(self, tasks: list[Task]) -> str:
        """Compute Blake3 or SHA256 integrity hash of execution plan."""
        task_data = json.dumps(
            [
                {
                    "task_id": t.task_id,
                    "dimension": t.dimension,
```

```
                "question_id": t.question_id,
                "policy_area": t.policy_area,
                "chunk_id": t.chunk_id,
            }
            for t in tasks
        ],
        sort_keys=True,
    ).encode("utf-8")

    if BLAKE3_AVAILABLE:
        return blake3.blake3(task_data).hexdigest()
    else:
        return hashlib.sha256(task_data).hexdigest()

def _construct_execution_plan_phase_8_4(
    self,
    sorted_tasks: list[Task],
    plan_id: str,
    chunk_count: int,
    question_count: int,
    integrity_hash: str,
) -> ExecutionPlan:
    """Phase 8.4: ExecutionPlan dataclass construction.

    Constructs the final execution artifact from the sorted task list produced in
    Phase 8.2, converting sorted_tasks to an immutable tuple, constructing a
    metadata dictionary with generation_timestamp (UTC ISO 8601),
    synchronizer_version "2.0.0", chunk_count from the chunk matrix,
    question_count and task_count, invoking the ExecutionPlan constructor with
    plan_id from Phase 8.3 and tasks_tuple with metadata_dict as keyword arguments,
    wrapping the constructor call in try-except to catch TypeError from dataclass
    validation and re-raise as ValueError with context-specific message, then
    verifying task order preservation by checking that all adjacent task_id pairs
    maintain lexicographic ordering and raising ValueError if any violation is
    detected before emitting an info-level structured log event and returning the
    constructed ExecutionPlan instance.

    Args:
        sorted_tasks: List of Task objects sorted by task_id (from Phase 8.2)
        plan_id: Plan identifier string (from Phase 8.3)
        chunk_count: Number of chunks in the document
        question_count: Number of questions in the questionnaire
        integrity_hash: Blake3 or SHA256 hash of the task list

    Returns:
        ExecutionPlan instance with validated task ordering

    Raises:
        ValueError: If dataclass validation fails or task ordering is violated
    """
    tasks_tuple = tuple(sorted_tasks)

    metadata_dict = {
        "generation_timestamp": time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime()),
```

```
            "synchronizer_version": "2.0.0",
            "chunk_count": chunk_count,
            "question_count": question_count,
            "task_count": len(tasks_tuple),
        }

        try:
            plan = ExecutionPlan(
                plan_id=plan_id,
                tasks=tasks_tuple,
                chunk_count=metadata_dict["chunk_count"],
                question_count=metadata_dict["question_count"],
                integrity_hash=integrity_hash,
                created_at=metadata_dict["generation_timestamp"],
                correlation_id=self.correlation_id,
            )
        except TypeError as e:
            raise ValueError(
                f"ExecutionPlan dataclass construction failed: {e}. "
                f"Constructor validation rejected arguments (plan_id={plan_id}, "
                f"task_count={len(tasks_tuple)}, chunk_count={chunk_count}, "
                f"question_count={question_count})"
            ) from e

        for i in range(len(tasks_tuple) - 1):
            current_task_id = tasks_tuple[i].task_id
            next_task_id = tasks_tuple[i + 1].task_id

            if current_task_id >= next_task_id:
                raise ValueError(
                    f"Task order preservation violation detected at index {i}: "
                    f"task_id '{current_task_id}' >= task_id '{next_task_id}'. "
                     f"Expected strict lexicographic ordering maintained after Phase 8.2
sort."
                )

        logger.info(
            json.dumps(
                {
                    "event": "execution_plan_phase_8_4_complete",
                    "plan_id": plan_id,
                    "task_count": len(tasks_tuple),
                    "chunk_count": chunk_count,
                    "question_count": question_count,
                    "integrity_hash": integrity_hash,
                    "synchronizer_version": metadata_dict["synchronizer_version"],
                    "generation_timestamp": metadata_dict["generation_timestamp"],
                    "correlation_id": self.correlation_id,
                    "phase": "execution_plan_construction_phase_8_4",
                }
            )
        )

        return plan
```

```
def _validate_cross_task_cardinality(
    self, plan: ExecutionPlan, questions: list[dict[str, Any]]
) -> None:
    """Validate cross-task cardinality and log task distribution statistics.

    Extracts unique chunk IDs from execution plan tasks, computes expected
    reference counts by filtering questions for matching policy_area_id and
    dimension_id (parsed from chunk_id), compares actual task counts per chunk
    against expected counts, and emits warning-level logs for mismatches.

    Also collects chunk usage statistics (mean, median, min, max) across all
    unique chunks, policy area task distribution mapping, and dimension coverage
    validation, culminating in a single info-level log entry with complete
    observability into task distribution patterns.

    Args:
        plan: ExecutionPlan containing all constructed tasks
        questions: List of original question dictionaries

    Raises:
        None - Discrepancies emit warnings but do not raise exceptions since
               they may reflect legitimate sparse coverage rather than errors
    """
    unique_chunks: set[str] = set()
    chunk_task_counts: dict[str, int] = {}

    for task in plan.tasks:
        chunk_id = task.chunk_id
        unique_chunks.add(chunk_id)
        chunk_task_counts[chunk_id] = chunk_task_counts.get(chunk_id, 0) + 1

    for chunk_id, actual_count in chunk_task_counts.items():
        try:
            parts = chunk_id.split("-")
            if len(parts) >= 2:
                policy_area_id = parts[0]
                dimension_id = parts[1]

                expected_count = sum(
                    1
                    for q in questions
                    if q.get("policy_area_id") == policy_area_id
                    and q.get("dimension_id") == dimension_id
                )

                if actual_count != expected_count:
                    logger.warning(
                        json.dumps(
                            {
                                "event": "cross_task_cardinality_mismatch",
                                "chunk_id": chunk_id,
                                "policy_area_id": policy_area_id,
                                "dimension_id": dimension_id,
```

```
                            "expected_count": expected_count,
                            "actual_count": actual_count,
                            "correlation_id": self.correlation_id,
                            "timestamp": time.time(),
                        }
                    )
                )
    except (IndexError, ValueError) as e:
        logger.warning(
            json.dumps(
                {
                    "event": "chunk_id_parse_error",
                    "chunk_id": chunk_id,
                    "error": str(e),
                    "correlation_id": self.correlation_id,
                    "timestamp": time.time(),
                }
            )
        )

chunk_counts = list(chunk_task_counts.values())
chunk_usage_stats: dict[str, float] = {}

if chunk_counts:
    chunk_usage_stats = {
        "mean": statistics.mean(chunk_counts),
        "median": statistics.median(chunk_counts),
        "min": float(min(chunk_counts)),
        "max": float(max(chunk_counts)),
    }

tasks_per_policy_area: dict[str, int] = {}
for task in plan.tasks:
    try:
        parts = task.chunk_id.split("-")
        if len(parts) >= 1:
            policy_area_id = parts[0]
            tasks_per_policy_area[policy_area_id] = (
                tasks_per_policy_area.get(policy_area_id, 0) + 1
            )
    except (IndexError, ValueError):
        pass

tasks_per_dimension: dict[str, int] = {}
for task in plan.tasks:
    try:
        parts = task.chunk_id.split("-")
        if len(parts) >= 2:
            dimension_id = parts[1]
            tasks_per_dimension[dimension_id] = (
                tasks_per_dimension.get(dimension_id, 0) + 1
            )
    except (IndexError, ValueError):
        pass
```

```
    logger.info(
        json.dumps(
            {
                "event": "cross_task_cardinality_validation_complete",
                "total_unique_chunks": len(unique_chunks),
                "tasks_per_policy_area": tasks_per_policy_area,
                "tasks_per_dimension": tasks_per_dimension,
                "chunk_usage_stats": chunk_usage_stats,
                "correlation_id": self.correlation_id,
                "timestamp": time.time(),
            }
        )
    )

@synchronization_duration.time()
def build_execution_plan(self) -> ExecutionPlan:
    """Build deterministic execution plan mapping questions to chunks.

    Uses validated chunk matrix if available, otherwise falls back to
    legacy document_chunks iteration mode.

    Returns:
        ExecutionPlan with deterministic plan_id and integrity_hash

    Raises:
        ValueError: If question data is invalid or chunk matrix lookup fails
    """
    if self.chunk_matrix is not None:
        return self._build_with_chunk_matrix()
    else:
        return self._build_with_legacy_chunks()

def _build_with_chunk_matrix(self) -> ExecutionPlan:
    """Build execution plan using validated chunk matrix.

    Orchestrates Phases 2-7 of irrigation synchronization:
    - Phase 0: JOIN table construction (if enabled)
    - Phase 2: Question extraction
    - Phase 3: Chunk routing (OBJECTIVE 3 INTEGRATION)
    - Phase 4: Pattern filtering (policy_area_id or contract-driven)
    - Phase 5: Signal resolution
    - Phase 6: Schema validation
    - Phase 7: Task construction

    Returns:
        ExecutionPlan with validated tasks

    Raises:
        ValueError: On routing failures, validation errors
    """
    logger.info(
        json.dumps(
            {
```

```
            "event": "task_construction_start",
            "correlation_id": self.correlation_id,
            "question_count": self.question_count,
            "chunk_count": self.chunk_count,
            "mode": "chunk_matrix",
            "join_table_enabled": self.enable_join_table,
            "phase": "synchronization_phase_2",
            "timestamp": time.time(),
        }
    )
)

# Phase 0: Build JOIN table if enabled
if self.enable_join_table and self.chunk_matrix:
    chunks = self.chunk_matrix._preprocessed_document.chunks
    self.join_table = self._build_join_table_if_enabled(chunks)

try:
    if self.question_count == 0:
        synchronization_failures.labels(error_type="empty_questions").inc()
        raise ValueError(
            "No questions extracted from questionnaire. "
            "Cannot build tasks with empty question set."
        )

    questions = self._extract_questions()

    if not questions:
        raise ValueError(
            "No questions extracted from questionnaire. "
            "Cannot build tasks with empty question set."
        )

    tasks: list[ExecutableTask] = []
    routing_successes = 0
    routing_failures = 0
    generated_task_ids: set[str] = set()

    for idx, question in enumerate(questions, start=1):
        question_id = question.get("question_id", f"UNKNOWN_{idx}")
        policy_area_id = question.get("policy_area_id", "UNKNOWN")
        dimension_id = question.get("dimension_id", "UNKNOWN")
        chunk_id = "UNKNOWN"

        try:
            routing_result = self.validate_chunk_routing(question)
            routing_successes += 1
            chunk_id = routing_result.chunk_id

            # Phase 4: Pattern filtering - contract-driven or generic
            if self.join_table and self.executor_contracts:
                # Contract-driven pattern irrigation (higher precision)
                contract = self._find_contract_for_question(question)
                if contract:
```

```
                                                                applicable_patterns  =
self._filter_patterns_from_contract(contract)
                        else:
                            # Fallback to generic if contract not found
                            logger.warning(
                                json.dumps(
                                    {
                                                                                "event":
"contract_not_found_fallback_to_generic",
                                        "question_id": question_id,
                                        "policy_area_id": policy_area_id,
                                        "dimension_id": dimension_id,
                                        "correlation_id": self.correlation_id,
                                    }
                                )
                            )
                            patterns_raw = question.get("patterns", [])
                            applicable_patterns = self._filter_patterns(
                                patterns_raw, routing_result.policy_area_id
                            )
                    else:
                        # Generic PA-level pattern filtering
                        patterns_raw = question.get("patterns", [])
                        applicable_patterns = self._filter_patterns(
                            patterns_raw, routing_result.policy_area_id
                        )

                    # Phase 5 validation: Ensure signal_registry initialized
                    if self.signal_registry is None:
                        raise ValueError(
                            f"SignalRegistry required for Phase 5 signal resolution "
                            f"but not initialized for question {question_id}"
                        )

                    resolved_signals = self._resolve_signals_for_question(
                        question,
                        routing_result.target_chunk,
                        self.signal_registry,
                    )

                    # Phase 6: Schema validation (four subphase pipeline)
                    # Validates structural compatibility and semantic constraints
                    # Allows TypeError/ValueError to propagate to outer handler
                    validate_phase6_schema_compatibility(
                        question=question,
                        chunk_expected_elements=routing_result.expected_elements,
                        chunk_id=routing_result.chunk_id,
                        policy_area_id=routing_result.policy_area_id,
                        correlation_id=self.correlation_id,
                    )

                    task = self._construct_task(
                        question,
                        routing_result,
```

```
                    applicable_patterns,
                    resolved_signals,
                    generated_task_ids,
                )
                tasks.append(task)

                if idx % 50 == 0:
                    logger.info(
                        json.dumps(
                            {
                                "event": "task_construction_progress",
                                "tasks_completed": idx,
                                "total_questions": len(questions),
                                "progress_pct": round(
                                    100 * idx / len(questions), 2
                                ),
                                "correlation_id": self.correlation_id,
                            }
                        )
                    )

            except (ValueError, TypeError) as e:
                routing_failures += 1

                logger.error(
                    json.dumps(
                        {
                            "event": "task_construction_failure",
                            "error_event": "routing_or_signal_failure",
                            "question_id": question_id,
                            "question_index": idx,
                            "policy_area_id": policy_area_id,
                            "dimension_id": dimension_id,
                            "chunk_id": chunk_id,
                            "error_type": type(e).__name__,
                            "error_message": str(e),
                            "correlation_id": self.correlation_id,
                            "timestamp": time.time(),
                        }
                    ),
                    exc_info=True,
                )

                raise

        expected_task_count = len(questions)
        actual_task_count = len(tasks)

        if actual_task_count != expected_task_count:
            raise ValueError(
                f"Task count mismatch: Expected {expected_task_count} tasks "
                f"but constructed {actual_task_count}. "
                                f"Routing  successes:  {routing_successes},  failures:
{routing_failures}"
```

```
    )

tasks, plan_id = self._assemble_execution_plan(
    tasks, questions, self.correlation_id
)

logger.info(
    json.dumps(
        {
            "event": "task_construction_complete",
            "total_tasks": actual_task_count,
            "routing_successes": routing_successes,
            "routing_failures": routing_failures,
            "success_rate": round(
                100 * routing_successes / max(expected_task_count, 1), 2
            ),
            "correlation_id": self.correlation_id,
            "timestamp": time.time(),
        }
    )
)

legacy_tasks = []
for task in tasks:
    legacy_task = Task(
        task_id=task.task_id,
        dimension=task.dimension_id,
        question_id=task.question_id,
        policy_area=task.policy_area_id,
        chunk_id=task.chunk_id,
        chunk_index=0,
        question_text="",
    )
    legacy_tasks.append(legacy_task)

integrity_hash = self._compute_integrity_hash(legacy_tasks)

plan = self._construct_execution_plan_phase_8_4(
    sorted_tasks=legacy_tasks,
    plan_id=plan_id,
    chunk_count=self.chunk_count,
    question_count=len(questions),
    integrity_hash=integrity_hash,
)

self._validate_cross_task_cardinality(plan, questions)

# Generate verification manifest if JOIN table was built
if self.join_table and SYNCHRONIZER_AVAILABLE:
    try:
        manifest = generate_verification_manifest(
            self.join_table,
            include_full_bindings=False  # Reduce size
        )
```

```
                    # Save manifest if path available
                    manifest_dir = Path("artifacts/manifests")
                    manifest_dir.mkdir(parents=True, exist_ok=True)
                                                        manifest_path   =   manifest_dir   /
"executor_chunk_synchronization_manifest.json"

                    save_verification_manifest(manifest, manifest_path)

                    logger.info(
                        json.dumps(
                            {
                                "event": "verification_manifest_generated",
                                "manifest_path": str(manifest_path),
                                "bindings_count": len(self.join_table),
                                "success": manifest.get("success", False),
                                "correlation_id": self.correlation_id,
                            }
                        )
                    )
                except Exception as e:
                    logger.warning(
                        json.dumps(
                            {
                                "event": "verification_manifest_generation_failed",
                                "error": str(e),
                                "correlation_id": self.correlation_id,
                            }
                        )
                    )

            logger.info(
                json.dumps(
                    {
                        "event": "build_execution_plan_complete",
                        "correlation_id": self.correlation_id,
                        "plan_id": plan_id,
                        "task_count": len(legacy_tasks),
                        "chunk_count": self.chunk_count,
                        "question_count": len(questions),
                        "integrity_hash": integrity_hash,
                        "chunk_matrix_validated": True,
                        "join_table_enabled": self.enable_join_table,
                          "join_table_bindings": len(self.join_table) if self.join_table
else 0,
                        "mode": "chunk_matrix",
                        "phase": "synchronization_phase_complete",
                    }
                )
            )

            return plan

        except ValueError as e:
```

```
            synchronization_failures.labels(error_type="validation_failure").inc()
            logger.error(
                json.dumps(
                    {
                        "event": "build_execution_plan_error",
                        "correlation_id": self.correlation_id,
                        "error": str(e),
                        "error_type": "validation_failure",
                    }
                )
            )
            raise
        except Exception as e:
            synchronization_failures.labels(error_type=type(e).__name__).inc()
            logger.error(
                json.dumps(
                    {
                        "event": "build_execution_plan_error",
                        "correlation_id": self.correlation_id,
                        "error": str(e),
                        "error_type": type(e).__name__,
                    }
                )
            )
            raise

    def _build_with_legacy_chunks(self) -> ExecutionPlan:
        """Build execution plan using legacy document_chunks list.

        DEPRECATED: This method is deprecated and will be removed in a future version.
           All consumers should migrate to using PreprocessedDocument with ChunkMatrix
validation.
            The legacy mode lacks the robust validation and deterministic routing of
ChunkMatrix.
        """
        import warnings
        warnings.warn(
            "Legacy chunk mode is deprecated and will be removed in a future version. "
            "Please migrate to PreprocessedDocument with ChunkMatrix validation.",
            DeprecationWarning,
            stacklevel=2
        )

        logger.warning(
            json.dumps(
                {
                    "event": "legacy_chunk_mode_deprecated",
                    "correlation_id": self.correlation_id,
                    "message": "Legacy chunk mode will be removed in future version",
                    "migration_guide": "Use PreprocessedDocument with ChunkMatrix",
                    "timestamp": time.time()
                }
            )
        )
```

```
logger.info(
    json.dumps(
        {
            "event": "build_execution_plan_start",
            "correlation_id": self.correlation_id,
            "question_count": self.question_count,
            "chunk_count": self.chunk_count,
            "mode": "legacy_chunks",
            "phase": "synchronization_phase_0",
        }
    )
)

try:
    if not self.document_chunks:
        synchronization_failures.labels(error_type="empty_chunks").inc()
        raise ValueError("No document chunks provided")

    if self.question_count == 0:
        synchronization_failures.labels(error_type="empty_questions").inc()
        raise ValueError("No questions found in questionnaire")

    questions = self._extract_questions()
    policy_areas = [f"PA{i:02d}" for i in range(1, 11)]

    tasks: list[Task] = []

    for question in questions:
        for policy_area in policy_areas:
            for chunk_idx, chunk in enumerate(self.document_chunks):
                chunk_id = chunk.get("chunk_id", f"chunk_{chunk_idx:04d}")

                task_id = f"{question['question_id']}_{policy_area}_{chunk_id}"

                task = Task(
                    task_id=task_id,
                    dimension=question["dimension"],
                    question_id=question["question_id"],
                    policy_area=policy_area,
                    chunk_id=chunk_id,
                    chunk_index=chunk_idx,
                    question_text=question["question_text"],
                )

                tasks.append(task)

                tasks_constructed.labels(
                    dimension=question["dimension"], policy_area=policy_area
                ).inc()

    sorted_tasks = sorted(tasks, key=lambda t: t.task_id)

    if len(sorted_tasks) != len(tasks):
```

```
                raise RuntimeError(
                                f"Task ordering corruption detected: sorted task count
{len(sorted_tasks)} "
                    f"does not match input task count {len(tasks)}"
                )

            task_serialization = [
                {
                    "task_id": t.task_id,
                    "question_id": t.question_id,
                    "dimension": t.dimension,
                    "policy_area": t.policy_area,
                    "chunk_id": t.chunk_id,
                }
                for t in sorted_tasks
            ]

            json_bytes = json.dumps(
                task_serialization, sort_keys=True, separators=(",", ":")
            ).encode("utf-8")

            plan_id = hashlib.sha256(json_bytes).hexdigest()

            if len(plan_id) != SHA256_HEX_DIGEST_LENGTH:
                raise ValueError(
                                 f"Plan identifier validation failure: expected length
{SHA256_HEX_DIGEST_LENGTH} but got {len(plan_id)}; "
                    "SHA256 implementation may be compromised or monkey-patched"
                )

            if not all(c in "0123456789abcdef" for c in plan_id):
                raise ValueError(
                     "Plan identifier validation failure: expected lowercase hexadecimal
but got "
                       "characters outside '0123456789abcdef' set; SHA256 implementation
may be "
                    "compromised or monkey-patched"
                )

            integrity_hash = self._compute_integrity_hash(sorted_tasks)

            plan = self._construct_execution_plan_phase_8_4(
                sorted_tasks=sorted_tasks,
                plan_id=plan_id,
                chunk_count=self.chunk_count,
                question_count=len(questions),
                integrity_hash=integrity_hash,
            )

            self._validate_cross_task_cardinality(plan, questions)

            logger.info(
                json.dumps(
                    {
```

```
                    "event": "build_execution_plan_complete",
                    "correlation_id": self.correlation_id,
                    "plan_id": plan_id,
                    "task_count": len(tasks),
                    "chunk_count": self.chunk_count,
                    "question_count": len(questions),
                    "integrity_hash": integrity_hash,
                    "mode": "legacy_chunks",
                    "phase": "synchronization_phase_complete",
                }
            )
        )

        return plan

    except Exception as e:
        synchronization_failures.labels(error_type=type(e).__name__).inc()
        logger.error(
            json.dumps(
                {
                    "event": "build_execution_plan_error",
                    "correlation_id": self.correlation_id,
                    "error": str(e),
                    "error_type": type(e).__name__,
                }
            )
        )
        raise

def _validate_cross_task_contamination(self, execution_plan: ExecutionPlan) -> None:
    """Build traceability mappings for task-chunk relationship queries.

    Constructs two bidirectional dictionaries enabling efficient task-chunk
    relationship queries and stores them in ExecutionPlan metadata:
    - task_chunk_mapping: Maps each task_id to its chunk_id (one-to-one)
    - chunk_task_mapping: Maps each chunk_id to list of task_ids (one-to-many)

    Args:
        execution_plan: ExecutionPlan to enrich with traceability mappings

    Returns:
        None (modifies execution_plan.metadata in place)
    """
    task_chunk_mapping = {t.task_id: t.chunk_id for t in execution_plan.tasks}

    chunk_task_mapping: dict[str, list[str]] = {}
    for t in execution_plan.tasks:
        chunk_task_mapping.setdefault(t.chunk_id, []).append(t.task_id)

    execution_plan.metadata["task_chunk_mapping"] = task_chunk_mapping
    execution_plan.metadata["chunk_task_mapping"] = chunk_task_mapping

def _resolve_signals_for_question(
    self,
```

```
        question: dict[str, Any],
        target_chunk: ChunkData,
        signal_registry: SignalRegistry,
    ) -> tuple[Any, ...]:
        """Resolve signals for a question from registry.

        Performs signal resolution with comprehensive validation:
        - Normalizes signal_requirements to empty list if missing/None
        - Calls signal_registry.get_signals_for_chunk with requirements
        - Validates return type is list (raises TypeError if None)
        - Validates each signal has required fields (signal_id, signal_type, content)
        - Detects missing required signals (HARD STOP with ValueError)
        - Detects and warns about duplicate signal types
        - Returns immutable tuple of resolved signals

        Args:
            question: Question dict with signal_requirements field
            target_chunk: Target ChunkData for signal resolution
                    signal_registry: Registry implementing get_signals_for_chunk(chunk,
requirements)

        Returns:
            Immutable tuple of resolved signals

        Raises:
            TypeError: If signal_registry returns non-list type
            ValueError: If signal missing required field or required signals not found
        """
        question_id = question.get("question_id", "UNKNOWN")
        chunk_id = getattr(target_chunk, "chunk_id", "UNKNOWN")

        # Normalize signal_requirements to empty list if missing or None
        signal_requirements = question.get("signal_requirements")
        if signal_requirements is None:
            signal_requirements = []
        elif not isinstance(signal_requirements, list):
            # If it's a dict or other type, extract as list if possible
            if isinstance(signal_requirements, dict):
                signal_requirements = list(signal_requirements.keys())
            else:
                signal_requirements = []

        # Call signal_registry.get_signals_for_chunk
        resolved_signals = signal_registry.get_signals_for_chunk(
            target_chunk, signal_requirements
        )

        # Validate return is list type (raise TypeError if None)
        if resolved_signals is None:
            raise TypeError(
                        f"SignalRegistry returned {type(None).__name__} for question
{question_id} "
                f"chunk {chunk_id}, expected list"
            )
```

```
        if not isinstance(resolved_signals, list):
            raise TypeError(
                 f"SignalRegistry returned {type(resolved_signals).__name__} for question
{question_id} "
                f"chunk {chunk_id}, expected list"
            )

        # Validate each signal has required fields
        required_fields = ["signal_id", "signal_type", "content"]
        for i, signal in enumerate(resolved_signals):
            for field in required_fields:
                # Try both attribute and dict access
                has_field = False
                try:
                    if hasattr(signal, field):
                        getattr(signal, field)
                        has_field = True
                except (AttributeError, KeyError):
                    pass

                if not has_field:
                    try:
                        if isinstance(signal, dict) and field in signal:
                            has_field = True
                    except (TypeError, KeyError):
                        pass

                if not has_field:
                    raise ValueError(
                               f"Signal at index {i} missing field {field} for question
{question_id}"
                    )

        # Extract signal_types into set
        signal_types = set()
        for signal in resolved_signals:
            # Try attribute access first, then dict access
            signal_type = None
            try:
                if hasattr(signal, "signal_type"):
                    signal_type = signal.signal_type
            except AttributeError:
                pass

            if signal_type is None:
                try:
                    if isinstance(signal, dict):
                        signal_type = signal["signal_type"]
                except (KeyError, TypeError):
                    pass

            if signal_type is not None:
                signal_types.add(signal_type)
```

```
# Compute missing signals
requirements_set = set(signal_requirements) if signal_requirements else set()
missing_signals = requirements_set - signal_types

# Raise ValueError if non-empty (HARD STOP)
if missing_signals:
    missing_sorted = sorted(missing_signals)
    raise ValueError(
        f"Synchronization Failure for MQC {question_id}: "
        f"Missing required signals {missing_sorted} for chunk {chunk_id}"
    )

# Detect duplicates
if len(resolved_signals) > len(signal_types):
    # Find duplicate types for logging
    type_counts: dict[Any, int] = {}
    for signal in resolved_signals:
        signal_type = None
        try:
            if hasattr(signal, "signal_type"):
                signal_type = signal.signal_type
        except AttributeError:
            pass

        if signal_type is None:
            try:
                if isinstance(signal, dict):
                    signal_type = signal["signal_type"]
            except (KeyError, TypeError):
                pass

        if signal_type is not None:
            type_counts[signal_type] = type_counts.get(signal_type, 0) + 1

    duplicate_types = [t for t, count in type_counts.items() if count > 1]

    logger.warning(
        "signal_resolution_duplicates",
        extra={
            "question_id": question_id,
            "chunk_id": chunk_id,
            "correlation_id": self.correlation_id,
            "duplicate_types": duplicate_types,
        },
    )

# Emit success log
logger.debug(
    "signal_resolution_success",
    extra={
        "question_id": question_id,
        "chunk_id": chunk_id,
        "correlation_id": self.correlation_id,
```

```
                "resolved_count": len(resolved_signals),
                "required_count": len(signal_requirements),
                "signal_types": list(signal_types),
            },
        )

        # Return tuple for immutability
        return tuple(resolved_signals)

    def _serialize_and_verify_plan(self, plan: ExecutionPlan) -> str:
        """Serialize ExecutionPlan and verify round-trip integrity.

        Serializes the execution plan to JSON, deserializes it back, reconstructs
        an ExecutionPlan instance, and validates that plan_id and task count match
        the original to ensure serialization is lossless.

        Args:
            plan: ExecutionPlan instance to serialize and verify

        Returns:
            Validated serialized JSON string ready for persistent storage

        Raises:
            ValueError: If plan_id mismatch or task count mismatch detected
        """
        plan_dict = plan.to_dict()
        serialized_json = json.dumps(plan_dict, sort_keys=True, separators=(",", ":"))

        deserialized_dict = json.loads(serialized_json)
        reconstructed_plan = ExecutionPlan.from_dict(deserialized_dict)

        if reconstructed_plan.plan_id != plan.plan_id:
            raise ValueError(
                f"Serialization verification failed: plan_id mismatch "
                f"(original={plan.plan_id}, reconstructed={reconstructed_plan.plan_id})"
            )

        original_task_count = len(plan.tasks)
        reconstructed_task_count = len(reconstructed_plan.tasks)

        if reconstructed_task_count != original_task_count:
            raise ValueError(
                f"Serialization verification failed: task count mismatch "
                                                    f"(original={original_task_count},
reconstructed={reconstructed_task_count})"
            )

        return serialized_json

    def _archive_to_storage(
        self,
        serialized_json: str,
        execution_plan: ExecutionPlan,
        base_dir: Path,
```

```
) -> ExecutionPlan:
    """Archive execution plan to storage with atomic index update and rollback.

    Constructs storage path as base_dir / 'execution_plans' / f'{plan_id}.json',
    writes serialized JSON with verification, and atomically updates index with
    rollback logic for orphaned files.

    Args:
        serialized_json: Serialized JSON string of execution plan
        execution_plan: ExecutionPlan instance to archive
        base_dir: Base directory path for storage

    Returns:
        Original ExecutionPlan instance unchanged

    Raises:
        ValueError: If write fails (re-raised from IOError)
        IOError: If write verification fails (content mismatch)
    """
    plan_id = execution_plan.plan_id
    storage_path = base_dir / "execution_plans" / f"{plan_id}.json"

    try:
        storage_path.parent.mkdir(parents=True, exist_ok=True)
    except IOError as e:
        raise ValueError(
            f"Failed to create parent directories for plan_id={plan_id}, "
            f"storage_path={storage_path}: {e}"
        ) from e

    try:
        storage_path.write_text(serialized_json, encoding="utf-8")
    except IOError as e:
        raise ValueError(
            f"Failed to write execution plan for plan_id={plan_id}, "
            f"storage_path={storage_path}: {e}"
        ) from e

    try:
        read_content = storage_path.read_text(encoding="utf-8")
        if read_content != serialized_json:
            storage_path.unlink()
            raise IOError(
                f"Write verification failed for plan_id={plan_id}, "
                f"storage_path={storage_path}: content mismatch after write"
            )
    except IOError as e:
        if storage_path.exists():
            storage_path.unlink()
        raise

    index_path = base_dir / "execution_plans" / "index.jsonl"
    index_entry = {
        "plan_id": plan_id,
```

```
            "storage_path": str(storage_path),
            "created_at": execution_plan.created_at,
            "task_count": len(execution_plan.tasks),
            "integrity_hash": execution_plan.integrity_hash,
            "correlation_id": execution_plan.correlation_id,
        }

        try:
            with open(index_path, "a", encoding="utf-8") as f:
                f.write(json.dumps(index_entry) + "\n")
        except IOError as e:
            if storage_path.exists():
                storage_path.unlink()
            raise ValueError(
                f"Failed to update index for plan_id={plan_id}, "
                f"storage_path={storage_path}: {e}"
            ) from e

        logger.info(
            "execution_plan_archived",
            extra={
                "event": "execution_plan_archived",
                "plan_id": plan_id,
                "storage_path": str(storage_path),
                "task_count": len(execution_plan.tasks),
                "integrity_hash": execution_plan.integrity_hash,
                "correlation_id": execution_plan.correlation_id,
                "created_at": execution_plan.created_at,
            },
        )

        return execution_plan


__all__ = [
    "IrrigationSynchronizer",
    "ExecutionPlan",
    "Task",
    "ChunkRoutingResult",
    "SignalRegistry",
]
```

```
src/farfan_pipeline/phases/Phase_two/json_files_phase_two/executor_contracts/contract_transfor

"""
Contract Transformation Engine
Implements full Tier 1/2/3 enhancements for executor contracts
"""
import copy
import hashlib
import json
from datetime import datetime


class ContractTransformer:
    """Transform contracts with epistemological depth and methodological rigor"""

    EPISTEMOLOGICAL_TEMPLATES = {
        'PDETMunicipalPlanAnalyzer._score_indicators': {
                'paradigm': 'Quantitative indicator analysis with municipal development
framework',
                 'ontological_basis': 'Product indicators exist as measurable outcomes
traceable to baseline data, targets, and verification sources in municipal planning
documents',
                'epistemological_stance': 'Empirical-analytical: Knowledge emerges from
systematic scoring of indicator completeness (baseline, target, source presence) against
structured criteria',
            'theoretical_framework': [
                    'Results-based management: Product indicators must link outputs to
verifiable evidence (Kusek & Rist, 2004)',
                             'PDET framework: Territorial development plans require
gender-disaggregated metrics with accountability mechanisms'
            ],
                'justification': 'Scoring indicators reveals institutional capacity to
operationalize gender commitments through measurable, verifiable product delivery'
        },
        'OperationalizationAuditor.audit_evidence_traceability': {
            'paradigm': 'Evidence chain verification with traceability analysis',
             'ontological_basis': 'Evidence exists as traceable artifacts linking claims
to sources through verifiable chains of custody',
              'epistemological_stance': 'Critical-analytical: Knowledge validity depends
on transparent provenance from claim to primary source',
            'theoretical_framework': [
                'Evidence-based policy: Claims require traceable evidence chains (Nutley
et al., 2007)',
                   'Audit theory: Traceability establishes epistemic warrant for policy
assertions'
            ],
               'justification': 'Auditing traceability exposes gaps between rhetorical
commitments and evidentiary foundations'
        },
        'CausalInferenceSetup.assign_probative_value': {
            'paradigm': 'Bayesian probative value assignment for causal inference',
                'ontological_basis': 'Evidence has varying probative strength based on
causal proximity, reliability, and inferential warrant',
                    'epistemological_stance': 'Bayesian-inferential: Evidence strength
```

```
quantified through posterior probabilities given causal structure',
            'theoretical_framework': [
                 'Bayesian epistemology: Probative value reflects evidential support for
causal hypotheses (Joyce, 2003)',
                  'Beach & Pedersen: Process tracing assigns probative weight based on
test type (hoop/smoking gun/doubly decisive)'
            ],
             'justification': 'Probative value assignment enables weighted aggregation of
evidence with differential epistemic strength'
        },
        'BeachEvidentialTest.apply_test_logic': {
            'paradigm': 'Process-tracing evidential tests (Beach & Pedersen framework)',
              'ontological_basis': 'Evidence can be classified by logical strength: hoop
tests (necessary), smoking gun tests (sufficient), doubly decisive tests (both)',
               'epistemological_stance': 'Quasi-experimental: Causal mechanisms inferred
through structured evidential tests eliminating rival hypotheses',
            'theoretical_framework': [
                   'Beach & Pedersen (2013): Process tracing uses evidential tests to
evaluate causal mechanisms',
                  'Van Evera (1997): Tests vary in uniqueness and certainty, affecting
inferential strength'
            ],
               'justification': 'Structured evidential tests provide rigorous basis for
causal claims in observational policy contexts'
        },
        'TextMiningEngine.diagnose_critical_links': {
            'paradigm': 'Critical text mining with causal link detection',
                'ontological_basis': 'Texts contain latent causal structures detectable
through linguistic patterns indicating causal relationships',
               'epistemological_stance': 'Empirical-interpretive: Knowledge about policy
mechanisms emerges from detecting linguistic markers of causality',
            'theoretical_framework': [
                      'Causal discourse analysis: Texts reveal causal beliefs through
linguistic constructions (Fairclough, 2003)',
                  'Theory of change reconstruction: Policy documents encode theories of
change extractable via causal link detection (Weiss, 1995)'
            ],
                   'justification': 'Diagnosing critical causal links reveals whether
policymakers understand causal pathways between gender inequalities and determinants'
        },
        'IndustrialPolicyProcessor._extract_metadata': {
            'paradigm': 'Metadata extraction for policy document characterization',
              'ontological_basis': 'Policy documents contain structured and unstructured
metadata revealing document provenance, scope, and administrative context',
             'epistemological_stance': 'Empirical-descriptive: Document metadata provides
contextual knowledge for interpretation and validation',
            'theoretical_framework': [
                 'Document analysis: Metadata enables systematic comparison and quality
assessment (Prior, 2003)',
                       'Administrative records theory: Metadata reveals institutional
production conditions'
            ],
                   'justification': 'Metadata extraction enables quality filtering and
contextualization of evidence by document characteristics'
```

```
        },
        'IndustrialPolicyProcessor._calculate_quality_score': {
            'paradigm': 'Multi-dimensional document quality assessment',
               'ontological_basis': 'Document quality is a composite construct spanning
completeness, consistency, traceability, and specification',
                'epistemological_stance': 'Evaluative-analytical: Quality emerges from
weighted aggregation of measurable quality dimensions',
            'theoretical_framework': [
                    'Information quality framework: Quality assessed via completeness,
accuracy, consistency dimensions (Wang & Strong, 1996)',
                        'Policy quality metrics: Operationalization quality predicts
implementation success'
            ],
             'justification': 'Quality scoring enables prioritization of high-reliability
evidence and flagging of low-quality sources'
        },
        'AdaptivePriorCalculator.generate_traceability_record': {
            'paradigm': 'Bayesian prior adaptation with full provenance tracking',
              'ontological_basis': 'Prior distributions encode background knowledge that
should adapt as evidence accumulates, with adaptation traced for reproducibility',
             'epistemological_stance': 'Bayesian-reflexive: Priors updated systematically
with transparent documentation of adaptation rationale',
            'theoretical_framework': [
                   'Adaptive Bayesian inference: Priors updated via empirical Bayes or
hierarchical modeling (Gelman et al., 2013)',
                  'Provenance tracking: Scientific reproducibility requires documented
lineage of analytical choices'
            ],
             'justification': 'Traceability records enable auditing of prior choices and
sensitivity analysis of posterior conclusions'
        }
    }

    TECHNICAL_APPROACHES = {
        'PDETMunicipalPlanAnalyzer._score_indicators': {
            'method_type': 'rule_based_scoring_with_pattern_matching',
                     'algorithm': 'Structured indicator completeness scoring across
baseline/target/source dimensions',
            'steps': [
                {
                    'step': 1,
                     'description': 'Extract indicator mentions using regex patterns for
product indicators (capacitadas, kits entregados, etc.)'
                },
                {
                    'step': 2,
                         'description': 'For each indicator, search context window (±50
tokens) for baseline markers (Línea Base, LB, Valor inicial)'
                },
                {
                    'step': 3,
                       'description': 'Search for target markers (Meta Cuatrienio, Meta,
Valor esperado) with numeric extraction'
                },
```

```
                {
                    'step': 4,
                     'description': 'Detect verification sources (Fuente de verificación,
Medio de verificación, listados de asistencia)'
                },
                {
                    'step': 5,
                         'description': 'Calculate completeness score: 1.0 if all three
elements present, 0.67 if two, 0.33 if one, 0.0 if zero'
                }
            ],
            'assumptions': [
                       'Indicators appear in proximity to their baseline/target/source
specifications',
                   'Standard Colombian planning terminology used (Meta Cuatrienio, Línea
Base, etc.)',
                'Context window of ±50 tokens captures most indicator metadata'
            ],
            'limitations': [
                       'Cannot detect implicit baselines or targets without linguistic
markers',
                'May miss indicators specified in tables or structured data formats',
                      'Assumes Spanish-language documents with Colombian public sector
conventions'
            ],
            'complexity': 'O(n*p) where n=document sentences, p=indicator patterns'
        },
        'OperationalizationAuditor.audit_evidence_traceability': {
            'method_type': 'chain_of_custody_validation',
            'algorithm': 'Backward tracing from claims to sources with gap detection',
            'steps': [
                {
                    'step': 1,
                       'description': 'Identify quantitative claims (numeric assertions
about gender indicators)'
                },
                {
                    'step': 2,
                        'description': 'Extract source citations (DANE, INML, Secretary
reports) within claim context'
                },
                {
                    'step': 3,
                               'description': 'Validate source authority (official entity,
publication year, dataset ID)'
                },
                {
                    'step': 4,
                        'description': 'Flag traceability gaps: claims without sources,
sources without validation, broken citation chains'
                },
                {
                    'step': 5,
                       'description': 'Generate traceability score: (validated_claims /
```

```
total_claims) * confidence_weight'
                }
            ],
            'assumptions': [
                'Claims and sources appear in same paragraph or adjacent paragraphs',
                'Standard citation patterns used (según DANE, fuente: X, datos de Y)'
            ],
            'limitations': [
                'Cannot validate source authenticity without external database lookup',
                'May miss indirect citations or footnote-based referencing'
            ],
            'complexity': 'O(n*c) where n=claims, c=citation patterns'
        },
        'CausalInferenceSetup.assign_probative_value': {
            'method_type': 'bayesian_weight_assignment',
            'algorithm': 'Posterior probability computation for evidential strength',
            'steps': [
                {
                    'step': 1,
                                'description': 'Classify evidence by type (statistical,
observational, testimonial)'
                },
                {
                    'step': 2,
                     'description': 'Assign prior probative weight based on evidence type
(statistical: 0.9, observational: 0.7, testimonial: 0.5)'
                },
                {
                    'step': 3,
                         'description': 'Adjust for source reliability (official sources
+0.1, academic +0.15, anecdotal -0.2)'
                },
                {
                    'step': 4,
                          'description': 'Compute posterior probative value via Bayesian
updating given document quality score'
                },
                {
                    'step': 5,
                       'description': 'Return ranked evidence with probative values for
weighted aggregation'
                }
            ],
            'assumptions': [
                'Prior probative weights reflect true evidential strength hierarchy',
                'Source reliability is independent of evidence type'
            ],
            'limitations': [
                    'Subjective priors for probative weights may not generalize across
contexts',
                'Independence assumptions may not hold for correlated evidence'
            ],
            'complexity': 'O(e) where e=evidence elements'
        },
```

```
        'BeachEvidentialTest.apply_test_logic': {
            'method_type': 'structured_process_tracing',
            'algorithm': 'Multi-test evidential logic with elimination inferencing',
            'steps': [
                {
                    'step': 1,
                     'description': 'Define mechanism hypothesis (e.g., "lack of baseline
data causes incomplete indicator specification")'
                },
                {
                    'step': 2,
                            'description': 'Apply hoop test: Is baseline data presence
necessary? If absent, mechanism rejected'
                },
                {
                    'step': 3,
                      'description': 'Apply smoking gun test: Is complete indicator spec
sufficient to confirm mechanism? If present, mechanism confirmed'
                },
                {
                    'step': 4,
                     'description': 'Aggregate test results: passed hoop + passed smoking
gun = strong confirmation, failed hoop = rejection'
                },
                {
                    'step': 5,
                      'description': 'Return mechanism confidence: doubly decisive (both
passed) = 0.9, smoking gun only = 0.7, hoop only = 0.4, neither = 0.1'
                }
            ],
            'assumptions': [
                'Mechanisms are discrete and testable via observable implications',
                   'Tests are correctly classified as hoop/smoking gun based on logical
strength'
            ],
            'limitations': [
                    'Requires pre-specified mechanism hypotheses, cannot discover novel
mechanisms',
                'Test classification subjective and context-dependent'
            ],
            'complexity': 'O(m*t) where m=mechanisms, t=tests per mechanism'
        },
        'TextMiningEngine.diagnose_critical_links': {
            'method_type': 'pattern_based_causal_link_extraction',
            'algorithm': 'Multi-pattern regex matching with context window analysis',
            'steps': [
                {
                    'step': 1,
                       'description': 'Identify causal connectors (porque, por lo tanto,
conduce a, genera, resulta en)'
                },
                {
                    'step': 2,
                     'description': 'Extract entities before connector (cause) and after
```

```
connector (effect) within ±30 token window'
                },
                {
                    'step': 3,
                        'description': 'Classify link criticality based on proximity to
gender indicators (VBG, autonomía, participación)'
                },
                {
                    'step': 4,
                         'description': 'Filter for critical links: criticality ? 0.7,
coherence score ? 0.6 (cause-effect semantic similarity)'
                },
                {
                    'step': 5,
                     'description': 'Return ranked critical links with cause-effect pairs
and confidence scores'
                }
            ],
            'assumptions': [
                'Causal language reflects causal understanding (not merely rhetorical)',
                       'Critical links mention gender-related outcomes within 30-token
proximity'
            ],
            'limitations': [
                'Cannot detect implicit causality without linguistic markers',
                   'May miss causal relationships expressed across distant sentences or
paragraphs'
            ],
            'complexity': 'O(n*p) where n=sentences, p=causal patterns'
        },
        'IndustrialPolicyProcessor._extract_metadata': {
            'method_type': 'structured_metadata_extraction',
            'algorithm': 'Pattern-based metadata field extraction with validation',
            'steps': [
                {
                    'step': 1,
                              'description': 'Extract document identifiers (plan name,
municipality, year) from title and header sections'
                },
                {
                    'step': 2,
                        'description': 'Detect policy area classification (gender, rural
development, etc.) via keyword matching'
                },
                {
                    'step': 3,
                      'description': 'Parse temporal scope (plan period: 2020-2023) from
common metadata fields'
                },
                {
                    'step': 4,
                             'description': 'Identify responsible entities (Secretarías,
alcaldía) from authorship and approval sections'
                },
```

```
                {
                    'step': 5,
                        'description': 'Validate metadata completeness and flag missing
critical fields (year, municipality)'
                }
            ],
            'assumptions': [
                  'Metadata fields appear in standard locations (headers, footers, first
pages)',
                'Standard Colombian municipal plan structure followed'
            ],
            'limitations': [
                'Cannot extract metadata from unstructured or non-standard documents',
                'May fail on scanned PDFs without OCR or poorly formatted text'
            ],
            'complexity': 'O(n) where n=document tokens (single-pass extraction)'
        },
        'IndustrialPolicyProcessor._calculate_quality_score': {
            'method_type': 'composite_quality_scoring',
               'algorithm': 'Weighted aggregation of quality dimensions with threshold
checks',
            'steps': [
                {
                    'step': 1,
                              'description': 'Calculate completeness: (present_fields /
required_fields) * 0.3'
                },
                {
                    'step': 2,
                     'description': 'Calculate consistency: 1 - (contradictions_detected
/ total_claims) * 0.25'
                },
                {
                    'step': 3,
                         'description': 'Calculate traceability: (claims_with_sources /
total_claims) * 0.25'
                },
                {
                    'step': 4,
                                              'description': 'Calculate specification:
(indicators_with_baseline_target / total_indicators) * 0.2'
                },
                {
                    'step': 5,
                              'description': 'Aggregate: quality_score = completeness +
consistency + traceability + specification (0-1 scale)'
                }
            ],
            'assumptions': [
                  'Quality dimensions are independent and equally important within their
weights',
                'Weighted formula reflects true document quality construct'
            ],
            'limitations': [
```

```
                'Weights subjectively chosen, may not match all use cases',
                        'Cannot assess semantic quality or logical coherence beyond
contradiction detection'
            ],
            'complexity': 'O(n) where n=document elements (claims, indicators, fields)'
        },
        'AdaptivePriorCalculator.generate_traceability_record': {
            'method_type': 'provenance_logging_with_bayesian_updating',
            'algorithm': 'Structured prior adaptation with full audit trail generation',
            'steps': [
                {
                    'step': 1,
                             'description': 'Initialize prior distribution from expert
elicitation or empirical data (e.g., Beta(2, 5) for baseline presence)'
                },
                {
                    'step': 2,
                      'description': 'Update prior to posterior using observed data via
Bayesian conjugate updating'
                },
                {
                    'step': 3,
                      'description': 'Log adaptation record: prior parameters, observed
data, posterior parameters, update rationale'
                },
                {
                    'step': 4,
                           'description': 'Generate provenance graph: link posterior to
evidence elements and methods contributing to update'
                },
                {
                    'step': 5,
                     'description': 'Return traceability record with full audit trail for
reproducibility and sensitivity analysis'
                }
            ],
            'assumptions': [
                     'Conjugate priors available for efficient updating (Beta-Binomial,
Normal-Normal, etc.)',
                'Prior choice transparent and justified via expert knowledge or data'
            ],
            'limitations': [
                        'Non-conjugate updates require MCMC or variational inference
(computationally expensive)',
                     'Prior sensitivity not automatically assessed, requires additional
sensitivity analysis'
            ],
                'complexity': 'O(d) where d=data points for posterior update (O(1) for
conjugate, O(n*iterations) for MCMC)'
        }
    }

    OUTPUT_INTERPRETATIONS = {
        'PDETMunicipalPlanAnalyzer._score_indicators': {
```

```
            'output_structure': {
                                'indicator_scores': 'List of dicts with {indicator_name,
baseline_present, target_present, source_present, completeness_score}',
                            'aggregate_score': 'Mean completeness across all indicators (0-1
scale)',
                    'missing_elements': 'List of indicators missing baseline/target/source'
            },
            'interpretation_guide': {
                    'high_completeness': '?0.8: Indicators fully specified with baseline,
target, and verification sources',
                        'medium_completeness': '0.5-0.79: Most indicators have 2/3 elements,
some gaps remain',
                    'low_completeness': '<0.5: Systematic gaps in indicator specification,
likely implementation challenges'
            },
            'actionable_insights': [
                        'If baseline missing: Cannot assess progress, need baseline data
collection',
                                'If targets missing: Unclear success criteria, need target
specification',
                    'If sources missing: Verification impossible, need source identification
and validation protocols'
            ]
        },
        'OperationalizationAuditor.audit_evidence_traceability': {
            'output_structure': {
                    'traced_claims': 'List of claims with {claim_text, source_citation,
source_valid, traceability_score}',
                    'traceability_gaps': 'Claims without sources or with invalid sources',
                        'aggregate_traceability': 'Percentage of claims with valid source
traceability'
            },
            'interpretation_guide': {
                    'high_traceability': '?0.8: Strong evidence chain, most claims traceable
to valid sources',
                    'medium_traceability': '0.5-0.79: Moderate gaps, some claims lack source
validation',
                                'low_traceability': '<0.5: Weak evidence foundation, many
unsubstantiated claims'
            },
            'actionable_insights': [
                    'If many gaps: Request source documentation from plan authors',
                    'If invalid sources: Flag for verification with official databases',
                        'If high traceability: Evidence foundation strong, can proceed with
confidence'
            ]
        },
        'CausalInferenceSetup.assign_probative_value': {
            'output_structure': {
                                'evidence_elements': 'List with {element_id, evidence_type,
source_reliability, prior_weight, posterior_probative_value}',
                    'high_value_evidence': 'Elements with probative_value ? 0.8',
                    'low_value_evidence': 'Elements with probative_value < 0.5'
            },
```

```
            'interpretation_guide': {
                'high_probative': '?0.8: Strong evidential warrant, suitable for primary
inferences',
                    'medium_probative': '0.5-0.79: Moderate strength, useful for
triangulation',
                'low_probative': '<0.5: Weak warrant, use cautiously or discard'
            },
            'actionable_insights': [
                'Prioritize high-probative evidence for causal claims',
                'Use medium-probative for corroboration, not primary inference',
                'Flag low-probative evidence for exclusion or sensitivity analysis'
            ]
        },
        'BeachEvidentialTest.apply_test_logic': {
            'output_structure': {
                'mechanism': 'Hypothesized causal mechanism being tested',
                'hoop_test_result': '{passed: bool, evidence: str, confidence: float}',
                    'smoking_gun_result': '{passed: bool, evidence: str, confidence:
float}',
                    'overall_confidence': 'Mechanism confidence (0-1) based on test
outcomes'
            },
            'interpretation_guide': {
                    'doubly_decisive': 'Both tests passed (0.9): Strong mechanism
confirmation',
                'smoking_gun_only': 'Sufficient but not necessary (0.7): Mechanism
plausible but not unique',
                'hoop_only': 'Necessary but not sufficient (0.4): Mechanism possible but
not confirmed',
                    'both_failed': 'Neither test passed (0.1): Mechanism rejected or
unsupported'
            },
            'actionable_insights': [
                'Doubly decisive: Accept mechanism, use for intervention design',
                'Smoking gun only: Consider alternative mechanisms with same sufficient
evidence',
                'Hoop only: Mechanism survives but needs additional evidence',
                'Both failed: Reject mechanism, explore alternative causal pathways'
            ]
        },
        'TextMiningEngine.diagnose_critical_links': {
            'output_structure': {
                        'critical_links': 'List of {cause_entity, effect_entity,
causal_connector, criticality_score, coherence_score, context}',
                    'gender_focused_links': 'Subset of links directly mentioning gender
outcomes (VBG, autonomía, participación)',
                'link_count': 'Total number of critical causal links detected'
            },
            'interpretation_guide': {
                'high_criticality': '?0.8: Link directly connects to gender inequality
outcomes',
                'medium_criticality': '0.5-0.79: Link relates to intermediate factors
(economic, social)',
                'low_criticality': '<0.5: Peripheral causal relationship'
```

```
            },
            'actionable_insights': [
                   'If few critical links: Diagnosis lacks causal depth, may be purely
descriptive',
                    'If many links but low criticality: Diagnosis discusses tangential
issues, not core gender inequalities',
                     'If high-criticality links present: Diagnosis demonstrates causal
understanding, good foundation for intervention'
            ]
        },
        'IndustrialPolicyProcessor._extract_metadata': {
            'output_structure': {
                'document_id': 'Unique identifier (plan name + municipality + year)',
                       'metadata_fields': 'Dict with {municipality, year, policy_area,
temporal_scope, responsible_entity, author}',
                'completeness': 'Percentage of required metadata fields present',
                'validation_flags': 'Missing or invalid fields requiring attention'
            },
            'interpretation_guide': {
                'complete_metadata': '100%: All required fields present and valid',
                'partial_metadata': '50-99%: Some fields missing, usable with caveats',
                  'incomplete_metadata': '<50%: Critical fields missing, document may be
unreliable'
            },
            'actionable_insights': [
                'Complete metadata: Document well-characterized, suitable for analysis',
                'Partial metadata: Use with caution, note limitations in reporting',
                  'Incomplete metadata: Request additional documentation or exclude from
analysis'
            ]
        },
        'IndustrialPolicyProcessor._calculate_quality_score': {
            'output_structure': {
                      'quality_score': 'Composite score (0-1) aggregating completeness,
consistency, traceability, specification',
                 'dimension_scores': 'Dict with {completeness, consistency, traceability,
specification} subscores',
                     'quality_flags': 'Specific issues detected (contradictions, missing
sources, incomplete indicators)'
            },
            'interpretation_guide': {
                   'high_quality': '?0.8: Document meets quality standards, suitable for
primary evidence',
                'medium_quality': '0.6-0.79: Acceptable quality with minor issues',
                'low_quality': '<0.6: Quality concerns, use cautiously or exclude'
            },
            'actionable_insights': [
                'High quality: Proceed with analysis, document is reliable',
                'Medium quality: Flag specific issues, use with caveats',
                'Low quality: Request document revision or seek alternative sources'
            ]
        },
        'AdaptivePriorCalculator.generate_traceability_record': {
            'output_structure': {
```

```
                'prior_distribution': 'Initial distribution (e.g., Beta(2, 5))',
                     'observed_data': 'Data used for updating (e.g., 8 successes in 10
trials)',
                'posterior_distribution': 'Updated distribution (e.g., Beta(10, 7))',
                     'adaptation_rationale': 'Reason for update (observed data, expert
input)',
                   'provenance_graph': 'Linkage from posterior to contributing evidence
elements'
            },
            'interpretation_guide': {
                    'strong_update': 'Large shift in posterior mean (>0.2): Data highly
informative',
                     'moderate_update': 'Moderate shift (0.05-0.2): Data provides some
information',
                    'weak_update': 'Small shift (<0.05): Data weakly informative, prior
dominates'
            },
            'actionable_insights': [
                     'Strong update: Data-driven inference, posterior reflects observed
patterns',
                'Moderate update: Balanced inference, both prior and data contribute',
                 'Weak update: Prior-dominated, consider collecting more data or refining
prior',
                    'Always check provenance graph for sensitivity to specific evidence
elements'
            ]
        }
    }

    def transform_q011_contract(self, contract: dict) -> dict:
        """Apply full transformation to Q011 contract"""
        contract = copy.deepcopy(contract)

        contract = self._fix_identity_schema_coherence(contract)
        contract = self._fix_method_assembly_alignment(contract)
        contract = self._fix_output_schema_required(contract)
        contract = self._expand_epistemological_foundation(contract)
        contract = self._enhance_methodological_depth(contract)
        contract = self._enrich_human_answer_structure(contract)
        contract = self._update_timestamp_and_hash(contract)

        return contract

    def _fix_identity_schema_coherence(self, contract: dict) -> dict:
        """Fix A1: Identity-Schema const field coherence"""
        identity = contract['identity']
        schema_props = contract['output_contract']['schema']['properties']

        schema_props['question_id']['const'] = identity['question_id']
        schema_props['policy_area_id']['const'] = identity['policy_area_id']
        schema_props['dimension_id']['const'] = identity['dimension_id']
        schema_props['question_global']['const'] = identity['question_global']
        schema_props['base_slot']['const'] = identity['base_slot']
```

```
        return contract

    def _fix_method_assembly_alignment(self, contract: dict) -> dict:
        """Fix A2: Method-Assembly provides-sources alignment"""
        methods = contract['method_binding']['methods']
        provides = [m['provides'] for m in methods]

        contract['evidence_assembly']['assembly_rules'][0]['sources'] = provides
           contract['evidence_assembly']['assembly_rules'][0]['description'] = f'Combine
evidence from {len(provides)} methods'

        return contract

    def _fix_output_schema_required(self, contract: dict) -> dict:
        """Fix A4: Ensure all required fields are in properties"""
        schema = contract['output_contract']['schema']
        required = schema['required']
        properties = schema['properties']

        for field in required:
            if field not in properties:
                properties[field] = {
                    'type': ['object', 'null'],
                    'additionalProperties': True
                }

        return contract

    def _expand_epistemological_foundation(self, contract: dict) -> dict:
        """Expand epistemological_foundation with question-specific paradigms"""
                                                                    methods         =
contract['output_contract']['human_readable_output']['methodological_depth']['methods']

        for method in methods:
            method_key = f"{method['class_name']}.{method['method_name']}"

            if method_key in self.EPISTEMOLOGICAL_TEMPLATES:
                                                method['epistemological_foundation']   =
self.EPISTEMOLOGICAL_TEMPLATES[method_key]
            else:
                method['epistemological_foundation'] = {
                    'paradigm': f'{method["class_name"]} analytical paradigm',
                    'ontological_basis': f'Analysis via {method_key}',
                    'epistemological_stance': 'Empirical-analytical approach',
                    'theoretical_framework': [
                         f'Method {method["method_name"]} implements structured analysis
for {contract["identity"]["base_slot"]}'
                    ],
                                        'justification':  f'This  method  contributes  to
{contract["identity"]["base_slot"]} analysis'
                }

        return contract
```

```
    def _enhance_methodological_depth(self, contract: dict) -> dict:
        """Enhance technical_approach with detailed steps and output_interpretation"""
                                                                        methods         =
contract['output_contract']['human_readable_output']['methodological_depth']['methods']

        for method in methods:
            method_key = f"{method['class_name']}.{method['method_name']}"

            if method_key in self.TECHNICAL_APPROACHES:
                method['technical_approach'] = self.TECHNICAL_APPROACHES[method_key]
            else:
                method['technical_approach'] = {
                    'method_type': 'analytical_processing',
                    'algorithm': f'{method_key} algorithm',
                    'steps': [
                        {'step': 1, 'description': f'Execute {method["method_name"]}'},
                        {'step': 2, 'description': 'Process results'},
                        {'step': 3, 'description': 'Return structured output'}
                    ],
                    'assumptions': ['Input data is preprocessed and valid'],
                    'limitations': ['Method-specific limitations apply'],
                    'complexity': 'O(n) where n=input size'
                }

            if method_key in self.OUTPUT_INTERPRETATIONS:
                                                    method['output_interpretation']   =
self.OUTPUT_INTERPRETATIONS[method_key]
            else:
                method['output_interpretation'] = {
                    'output_structure': {
                        'result': f'Structured output from {method["method_name"]}'
                    },
                    'interpretation_guide': {
                        'high_confidence': '?0.8: Strong evidence',
                        'medium_confidence': '0.5-0.79: Moderate evidence',
                        'low_confidence': '<0.5: Weak evidence'
                    },
                    'actionable_insights': [
                        f'Use {method["method_name"]} results for downstream analysis'
                    ]
                }

        return contract

    def _enrich_human_answer_structure(self, contract: dict) -> dict:
        """Enrich concrete_example with granular validation against expected_elements"""
        has = contract['human_answer_structure']
        expected = contract['question_context']['expected_elements']

        validation_section = {}
        for elem in expected:
            elem_type = elem['type']
            elem_required = elem.get('required', False)
```

```
            concrete = has['concrete_example']
            elements_found = concrete.get('elements_found', [])
            matching = [e for e in elements_found if e.get('type') == elem_type]

            validation_section[elem_type] = {
                'required': elem_required,
                'found_in_example': len(matching) > 0,
                'count': len(matching)
            }

            if matching:
                                validation_section[elem_type]['example_element_id']  =
matching[0].get('element_id')

        has['validation_against_expected_elements'] = validation_section
        has['validation_against_expected_elements']['overall_validation_result'] = (
            'PASS - All required elements present'
                if all(v['found_in_example'] for k, v in validation_section.items() if
v['required'])
            else 'FAIL - Some required elements missing'
        )

        return contract

    def _update_timestamp_and_hash(self, contract: dict) -> dict:
        """Update contract metadata with new timestamp and hash"""
        contract['identity']['created_at'] = datetime.now().isoformat() + '+00:00'

            hashable = {k: v for k, v in contract.items() if k not in ['identity',
'calibration']}
        contract_str = json.dumps(hashable, sort_keys=True)
                                            contract['identity']['contract_hash']     =
hashlib.sha256(contract_str.encode()).hexdigest()

        return contract


def transform_contract_file(input_path: str, output_path: str) -> None:
    """Transform contract file and save to output"""
    with open(input_path) as f:
        contract = json.load(f)

    transformer = ContractTransformer()
    transformed = transformer.transform_q011_contract(contract)

    with open(output_path, 'w') as f:
        json.dump(transformed, f, indent=2, ensure_ascii=False)
```

src/farfan_pipeline/phases/Phase_two/json_files_phase_two/executor_contracts/cqvr_validator.py

```
"""
CQVR Contract Quality Validation and Remediation System
Implements Tier 1/2/3 validation rubric for executor contracts
"""
import json
from typing import Any


class CQVRValidator:
    """Contract Quality Validation with Rubric-based scoring (CQVR v2.0)"""

    def __init__(self):
        self.tier1_weight = 55
        self.tier2_weight = 30
        self.tier3_weight = 15
        self.total_weight = 100

    def validate_contract(self, contract: dict) -> dict[str, Any]:
        """Run full CQVR validation and return detailed report"""
        scores = {
            'A1_identity_schema': self._verify_identity_schema_coherence(contract),
            'A2_method_assembly': self._verify_method_assembly_alignment(contract),
            'A3_signal_integrity': self._verify_signal_requirements(contract),
            'A4_output_schema': self._verify_output_schema(contract),
            'B1_pattern_coverage': self._verify_pattern_coverage(contract),
            'B2_method_specificity': self._verify_method_specificity(contract),
            'B3_validation_rules': self._verify_validation_rules(contract),
            'C1_documentation': self._verify_documentation_quality(contract),
            'C2_human_template': self._verify_human_template(contract),
            'C3_metadata': self._verify_metadata_completeness(contract)
        }

        tier1_score = (scores['A1_identity_schema'] + scores['A2_method_assembly'] +
                       scores['A3_signal_integrity'] + scores['A4_output_schema'])
        tier2_score = (scores['B1_pattern_coverage'] + scores['B2_method_specificity'] +
                       scores['B3_validation_rules'])
        tier3_score = (scores['C1_documentation'] + scores['C2_human_template'] +
                       scores['C3_metadata'])

        total_score = tier1_score + tier2_score + tier3_score
        percentage = (total_score / self.total_weight) * 100

          triage_decision = self._triage_decision(tier1_score, tier2_score, total_score,
scores)

        return {
            'total_score': total_score,
            'percentage': percentage,
            'tier1_score': tier1_score,
            'tier2_score': tier2_score,
            'tier3_score': tier3_score,
            'breakdown': scores,
```

```
            'triage_decision': triage_decision,
            'passed': percentage >= 80 and tier1_score >= 45
        }

    def _verify_identity_schema_coherence(self, contract: dict) -> int:
        """A1: Identity-Schema coherence (20 pts)"""
        score = 0
        id_map = {
            'question_id': 5,
            'policy_area_id': 5,
            'dimension_id': 5,
            'question_global': 3,
            'base_slot': 2
        }

        identity = contract.get('identity', {})
                    schema_props   =   contract.get('output_contract',   {}).get('schema',
{}).get('properties', {})

        for field, points in id_map.items():
            identity_val = identity.get(field)
            schema_const = schema_props.get(field, {}).get('const')
            if identity_val == schema_const:
                score += points

        return score

    def _verify_method_assembly_alignment(self, contract: dict) -> int:
        """A2: Method-Assembly alignment (20 pts)"""
        methods = contract.get('method_binding', {}).get('methods', [])
        provides = {m['provides'] for m in methods}

        sources = set()
        for rule in contract.get('evidence_assembly', {}).get('assembly_rules', []):
            for src in rule.get('sources', []):
                if '*' not in src:
                    sources.add(src)

        orphan_sources = sources - provides
        if orphan_sources:
            orphan_penalty = min(10, len(orphan_sources) * 2.5)
            return max(0, int(20 - orphan_penalty))

        unused_ratio = len(provides - sources) / len(provides) if provides else 0
        usage_score = 5 * (1 - unused_ratio)

                method_count_ok  =  3  if  contract['method_binding']['method_count']  ==
len(methods) else 0

        return int(10 + usage_score + method_count_ok + 2)

    def _verify_signal_requirements(self, contract: dict) -> int:
        """A3: Signal requirements integrity (10 pts)"""
        reqs = contract.get('signal_requirements', {})
```

```
         if reqs.get('mandatory_signals') and reqs.get('minimum_signal_threshold', 0) <=
0:
            return 0

        score = 5

        valid_aggregations = ['weighted_mean', 'max', 'min', 'product', 'voting']
        if reqs.get('signal_aggregation') in valid_aggregations:
            score += 2

        mandatory = reqs.get('mandatory_signals', [])
          if all(isinstance(s, str) and '_' in s for s in mandatory) if mandatory else
True:
            score += 3

        return score

    def _verify_output_schema(self, contract: dict) -> int:
        """A4: Output schema validation (5 pts)"""
        schema = contract.get('output_contract', {}).get('schema', {})
        required = set(schema.get('required', []))
        properties = set(schema.get('properties', {}).keys())

        if required.issubset(properties):
            return 5

        missing = required - properties
        penalty = len(missing)
        return max(0, 5 - penalty)

    def _verify_pattern_coverage(self, contract: dict) -> int:
        """B1: Pattern coverage (10 pts)"""
        patterns = contract.get('question_context', {}).get('patterns', [])
        expected = contract.get('question_context', {}).get('expected_elements', [])

        if not patterns:
            return 5

        weights_valid = all(0 < p.get('confidence_weight', 0) <= 1 for p in patterns)
        weight_score = 3 if weights_valid else 0

        ids = [p.get('id') for p in patterns]
         id_score = 2 if len(ids) == len(set(ids)) and all(id and 'PAT-' in id for id in
ids) else 0

        return 5 + weight_score + id_score

    def _verify_method_specificity(self, contract: dict) -> int:
        """B2: Methodological specificity (10 pts)"""
             methods  =  contract.get('output_contract',  {}).get('human_readable_output',
{}).get(
            'methodological_depth', {}).get('methods', [])
```

```
        if not methods:
            return 5

        generic_phrases = ["Execute", "Process results", "Return structured output"]
        total_steps = 0
        non_generic_steps = 0

        for method in methods[:5]:
            steps = method.get('technical_approach', {}).get('steps', [])
            total_steps += len(steps)
            for s in steps:
                if isinstance(s, dict):
                    desc = s.get('description', '')
                else:
                    desc = str(s)
                if not any(g in desc for g in generic_phrases):
                    non_generic_steps += 1

        if total_steps == 0:
            return 5

        specificity_ratio = non_generic_steps / total_steps
        return int(10 * specificity_ratio)

    def _verify_validation_rules(self, contract: dict) -> int:
        """B3: Validation rules (10 pts)"""
        rules = contract.get('validation_rules', {}).get('rules', [])
        expected = contract.get('question_context', {}).get('expected_elements', [])

        if not rules:
            return 5

         failure_score = 2 if contract.get('error_handling', {}).get('failure_contract',
{}).get('emit_code') else 0

        return 5 + 3 + failure_score

    def _verify_documentation_quality(self, contract: dict) -> int:
        """C1: Documentation quality (5 pts)"""
        return 3

    def _verify_human_template(self, contract: dict) -> int:
        """C2: Human-readable template (5 pts)"""
            template = contract.get('output_contract', {}).get('human_readable_output',
{}).get('template', {})

        score = 0
        question_id = contract.get('identity', {}).get('question_id', '')
        if question_id in str(template.get('title', '')):
            score += 3

        if '{score}' in str(template) and '{evidence' in str(template):
            score += 2
```

```
    return score

def _verify_metadata_completeness(self, contract: dict) -> int:
    """C3: Metadata completeness (5 pts)"""
    identity = contract.get('identity', {})
    score = 0

    if identity.get('contract_hash') and len(identity['contract_hash']) == 64:
        score += 2
    if identity.get('created_at'):
        score += 1
    if identity.get('validated_against_schema'):
        score += 1
    if identity.get('contract_version') and '.' in identity['contract_version']:
        score += 1

    return score

def _triage_decision(self, tier1: int, tier2: int, total: int, scores: dict) -> str:
    """Determine triage decision"""
    if tier1 < 35:
        return self._reformulate_decision(scores)
    elif tier1 >= 45 and total >= 70:
        return "PARCHEAR_MINOR"
    elif tier1 >= 35 and total >= 60:
        return "PARCHEAR_MAJOR"
    else:
        return self._analyze_borderline(scores)

def _reformulate_decision(self, scores: dict) -> str:
    """Analyze when Tier 1 fails"""
    blockers = []

    if scores['A1_identity_schema'] < 15:
        blockers.append("IDENTITY_SCHEMA_MISMATCH")
    if scores['A2_method_assembly'] < 12:
        blockers.append("ASSEMBLY_SOURCES_BROKEN")
    if scores['A3_signal_integrity'] < 5:
        blockers.append("SIGNAL_THRESHOLD_ZERO")
    if scores['A4_output_schema'] < 3:
        blockers.append("SCHEMA_INVALID")

    if len(blockers) >= 2:
        return f"REFORMULAR_COMPLETO: {blockers}"
    elif "ASSEMBLY_SOURCES_BROKEN" in blockers:
        return "REFORMULAR_ASSEMBLY"
    elif "IDENTITY_SCHEMA_MISMATCH" in blockers:
        return "REFORMULAR_SCHEMA"
    else:
        return "PARCHEAR_CRITICO"

def _analyze_borderline(self, scores: dict) -> str:
    """Analyze borderline cases"""
    if scores['B2_method_specificity'] < 3:
```

```
            return "PARCHEAR_DOCS"
        if scores['B1_pattern_coverage'] < 6:
            return "PARCHEAR_PATTERNS"
        return "PARCHEAR_MAJOR"


class ContractRemediation:
    """Apply structural corrections to contracts"""

    def apply_structural_corrections(self, contract: dict) -> dict:
        """Apply all structural corrections"""
        contract = self._fix_identity_schema_coherence(contract)
        contract = self._fix_method_assembly_alignment(contract)
        contract = self._fix_output_schema_required(contract)
        return contract

    def _fix_identity_schema_coherence(self, contract: dict) -> dict:
        """Ensure identity and output_schema const fields match"""
        identity = contract.get('identity', {})
                    schema_props   =   contract.get('output_contract',   {}).get('schema',
{}).get('properties', {})

                    for   field   in   ['question_id',   'policy_area_id',   'dimension_id',
'question_global', 'base_slot']:
            identity_val = identity.get(field)
            if identity_val is not None:
                if field in schema_props:
                    schema_props[field]['const'] = identity_val

        return contract

    def _fix_method_assembly_alignment(self, contract: dict) -> dict:
        """Align assembly sources with method provides"""
        methods = contract.get('method_binding', {}).get('methods', [])
        provides = [m['provides'] for m in methods]

        assembly_rules = contract.get('evidence_assembly', {}).get('assembly_rules', [])
        if assembly_rules:
            assembly_rules[0]['sources'] = provides

        return contract

    def _fix_output_schema_required(self, contract: dict) -> dict:
        """Ensure all required fields are defined in properties"""
        schema = contract.get('output_contract', {}).get('schema', {})
        required = schema.get('required', [])
        properties = schema.get('properties', {})

        for field in required:
            if field not in properties:
                properties[field] = {
                    'type': 'object',
                    'additionalProperties': True
                }
```

```
        return contract


def validate_and_triage_contract(contract_path: str) -> tuple[dict, str]:
    """Validate contract and return report + triage decision"""
    with open(contract_path) as f:
        contract = json.load(f)

    validator = CQVRValidator()
    report = validator.validate_contract(contract)

    return report, report['triage_decision']
```

src/farfan_pipeline/phases/Phase_two/json_files_phase_two/executor_contracts/transform_q011.py

```
#!/usr/bin/env python3
"""
Q011 Contract Transformation Script
Applies full Tier 1/2/3 validation and transformation to Q011.v3.json
"""
import json
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).parent.parent.parent.parent.parent))

from
src.canonic_phases.Phase_two.json_files_phase_two.executor_contracts.contract_transforme
r import (
    ContractTransformer,
)
from  src.canonic_phases.Phase_two.json_files_phase_two.executor_contracts.cqvr_validator
import (
    ContractRemediation,
    CQVRValidator,
)


def main():
    """Execute Q011 transformation pipeline"""

    base_path = Path(__file__).parent / 'specialized'
    input_path = base_path / 'Q011.v3.json'
    output_path = base_path / 'Q011.v3.transformed.json'

    print("=" * 80)
    print("Q011 CONTRACT TRANSFORMATION PIPELINE")
    print("=" * 80)

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

    print(f"\n1. LOADING CONTRACT: {input_path.name}")
    print(f"   Question: {contract['question_context']['question_text'][:80]}...")

    print("\n2. RUNNING CQVR VALIDATOR (PRE-TRANSFORMATION)")
    validator = CQVRValidator()
    pre_report = validator.validate_contract(contract)

    print("\n   PRE-TRANSFORMATION SCORES:")
                print(f"           -    Total    Score:    {pre_report['total_score']}/100
({pre_report['percentage']:.1f}%)")
    print(f"   - Tier 1 (Critical): {pre_report['tier1_score']}/55")
    print(f"   - Tier 2 (Functional): {pre_report['tier2_score']}/30")
    print(f"   - Tier 3 (Quality): {pre_report['tier3_score']}/15")
    print(f"   - Triage Decision: {pre_report['triage_decision']}")
```

```
    print("\n   DETAILED BREAKDOWN:")
    for criterion, score in pre_report['breakdown'].items():
        max_scores = {
            'A1_identity_schema': 20, 'A2_method_assembly': 20,
            'A3_signal_integrity': 10, 'A4_output_schema': 5,
            'B1_pattern_coverage': 10, 'B2_method_specificity': 10,
            'B3_validation_rules': 10, 'C1_documentation': 5,
            'C2_human_template': 5, 'C3_metadata': 5
        }
        max_score = max_scores.get(criterion, 10)
        status = "?" if score >= max_score * 0.8 else "?"
        print(f"   {status} {criterion}: {score}/{max_score}")

    print("\n3. APPLYING STRUCTURAL CORRECTIONS")
    remediation = ContractRemediation()
    contract = remediation.apply_structural_corrections(contract)
    print("   - Fixed identity-schema coherence")
    print("   - Fixed method-assembly alignment")
    print("   - Fixed output_schema required fields")

    print("\n4. APPLYING TRANSFORMATIONS")
    transformer = ContractTransformer()
    contract = transformer.transform_q011_contract(contract)
    print("   - Expanded epistemological_foundation with question-specific paradigms")
    print("   - Enhanced methodological_depth with technical_approach steps")
    print("   - Enriched human_answer_structure with granular validation")
    print("   - Updated timestamp and contract hash")

    print("\n5. RUNNING CQVR VALIDATOR (POST-TRANSFORMATION)")
    post_report = validator.validate_contract(contract)

    print("\n   POST-TRANSFORMATION SCORES:")
               print(f"           -    Total    Score:    {post_report['total_score']}/100
({post_report['percentage']:.1f}%)")
    print(f"   - Tier 1 (Critical): {post_report['tier1_score']}/55")
    print(f"   - Tier 2 (Functional): {post_report['tier2_score']}/30")
    print(f"   - Tier 3 (Quality): {post_report['tier3_score']}/15")
    print(f"   - Triage Decision: {post_report['triage_decision']}")
    print(f"   - PASSED: {'YES ?' if post_report['passed'] else 'NO ?'}")

    print("\n   IMPROVEMENTS:")
    for criterion in pre_report['breakdown'].keys():
        pre_score = pre_report['breakdown'][criterion]
        post_score = post_report['breakdown'][criterion]
        delta = post_score - pre_score
        if delta > 0:
            print(f"   ? {criterion}: +{delta} pts ({pre_score} ? {post_score})")
        elif delta < 0:
            print(f"   ? {criterion}: {delta} pts ({pre_score} ? {post_score})")

    total_delta = post_report['total_score'] - pre_report['total_score']
     print(f"\n    TOTAL IMPROVEMENT: +{total_delta} pts ({pre_report['percentage']:.1f}%
? {post_report['percentage']:.1f}%)")
```

```
    print(f"\n6. SAVING TRANSFORMED CONTRACT: {output_path.name}")
    with open(output_path, 'w') as f:
        json.dump(contract, f, indent=2, ensure_ascii=False)

    print("\n7. VALIDATION SUMMARY")
    if post_report['passed']:
        print("   ? CONTRACT MEETS CQVR STANDARDS (?80/100, Tier 1 ?45)")
        print("   ? Ready for production deployment")
    else:
        print("   ? CONTRACT DOES NOT MEET CQVR STANDARDS")
        print("   - Required: ?80/100 total, ?45/55 Tier 1")
                print(f"       -  Achieved:  {post_report['total_score']}/100  total,
{post_report['tier1_score']}/55 Tier 1")
        print(f"   - Recommendation: {post_report['triage_decision']}")

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

    return 0 if post_report['passed'] else 1


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

src/farfan_pipeline/phases/Phase_two/phase6_validation.py

"""Phase 6: Schema Validation Pipeline - Four Subphase Architecture.

This module implements Phase 6 as a complete validation pipeline with four subphases:

Phase 6.1: Classification & Extraction
    - Extracts question_global via bracket notation (question["question_global"])
    - Extracts expected_elements via get method with None handling
    - Classifies types using isinstance checks in None-list-dict-invalid order
    - Stores classification tuple before any iteration occurs

Phase 6.2: Structural Validation
    - Checks invalid types first with human-readable type names
    - Enforces homogeneity allowing None compatibility
    - Validates list length equality and dict key set equality
    - Uses symmetric difference computation for dict key validation
    - Returns silently on dual-None without logging

Phase 6.3: Semantic Validation
    - Iterates deterministically via enumerate-zip for lists and sorted keys for dicts
    - Extracts type-required-minimum fields with get defaults
    - Implements asymmetric required implication as not-q-or-c boolean expression
    - Enforces c-min-greater-equal-q-min threshold ordering
    - Returns validated element count

Phase 6.4: Orchestrator
    - Invokes structural then semantic layers in sequence
    - Captures element count return value
    - Emits debug log with has_required_fields and has_minimum_thresholds computed
      via any-element-iteration
    - Logs info warning for None chunk schema with non-None question schema
    - Integrates into build_with_chunk_matrix loop after Phase 5 before construct_task
      - Allows TypeError-ValueError propagation to outer handler without try-except
wrapping

Architecture:
    Phase 6.1 ? Phase 6.2 ? Phase 6.3 ? Phase 6.4
    (Sequential root) (Structural) (Semantic) (Synchronization barrier)

Parallelization:
    - Phase 6.1: Sequential root (extracts and classifies)
    - Phase 6.2-6.3: Concurrency potential (independent validation layers)
    - Phase 6.4: Synchronization barrier (aggregates results)
"""

from __future__ import annotations

import logging
from typing import Any

logger = logging.getLogger(__name__)

MAX_QUESTION_GLOBAL = 999

```
def _classify_expected_elements_type(value: Any) -> str:  # noqa: ANN401
    """Phase 6.1: Classify expected_elements type using isinstance checks.

    Performs type classification in None-list-dict-invalid order via identity
    test for None, then isinstance checks for list and dict, with any other
    type classified as invalid.

    Args:
        value: Value to classify (expected_elements from question or chunk)

    Returns:
        Type classification string: "none", "list", "dict", or "invalid"

    Classification Order:
        1. None via identity test (value is None)
        2. list via isinstance(value, list)
        3. dict via isinstance(value, dict)
        4. invalid for all other types
    """
    if value is None:
        return "none"
    elif isinstance(value, list):
        return "list"
    elif isinstance(value, dict):
        return "dict"
    else:
        return "invalid"


def _extract_and_classify_schemas(
    question: dict[str, Any],
    chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
    question_id: str,
) -> tuple[int, Any, Any, str, str]:  # noqa: ANN401
    """Phase 6.1: Extract question_global and expected_elements, classify types.

    Extracts question_global via bracket notation (question["question_global"])
    and expected_elements via get method with None default. Classifies both
    schema types and stores classification tuple before any iteration occurs.

    Args:
        question: Question dictionary from questionnaire
        chunk_expected_elements: expected_elements from chunk routing result
        question_id: Question identifier for error reporting

    Returns:
        Tuple of (question_global, question_schema, chunk_schema,
                  question_type, chunk_type)

    Raises:
        ValueError: If question_global is missing, invalid type, or out of range
    """
```

```
    # Extract question_global via bracket notation
    try:
        question_global = question["question_global"]
    except KeyError as e:
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            "question_global field is required but missing"
        ) from e

    # Validate question_global
    if not isinstance(question_global, int):
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            f"question_global must be an integer, got {type(question_global).__name__}"
        )

    if not (0 <= question_global <= MAX_QUESTION_GLOBAL):
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
             f"question_global must be between 0 and {MAX_QUESTION_GLOBAL} inclusive, got
{question_global}"
        )

    # Extract expected_elements via get method with None handling
    question_schema = question.get("expected_elements")
    chunk_schema = chunk_expected_elements

    # Classify types using isinstance checks in None-list-dict-invalid order
    question_type = _classify_expected_elements_type(question_schema)
    chunk_type = _classify_expected_elements_type(chunk_schema)

    # Store classification tuple before any iteration occurs
    return question_global, question_schema, chunk_schema, question_type, chunk_type


def _validate_structural_compatibility(
    question_schema: Any,  # noqa: ANN401
    chunk_schema: Any,  # noqa: ANN401
    question_type: str,
    chunk_type: str,
    question_id: str,
    correlation_id: str,
) -> None:
    """Phase 6.2: Validate structural compatibility with type homogeneity checks.

    Checks invalid types first with human-readable type names, enforces
    homogeneity allowing None compatibility, validates list length equality
    and dict key set equality with symmetric difference computation, and
    returns silently on dual-None without logging.

    Args:
        question_schema: expected_elements from question
        chunk_schema: expected_elements from chunk
        question_type: Classified type of question schema
```

```
    chunk_type: Classified type of chunk schema
    question_id: Question identifier for error messages
    correlation_id: Correlation ID for distributed tracing

Raises:
    TypeError: If either schema has invalid type (not list, dict, or None)
    ValueError: If schemas have heterogeneous types (not allowing None),
                list length mismatch, or dict key set mismatch

Returns:
    None (returns silently on dual-None or successful validation)
"""
# Check invalid types first with human-readable type names
if question_type == "invalid":
    raise TypeError(
        f"Schema validation failure for question {question_id}: "
        f"expected_elements from question has invalid type "
        f"{type(question_schema).__name__}, expected list, dict, or None "
        f"[correlation_id={correlation_id}]"
    )

if chunk_type == "invalid":
    raise TypeError(
        f"Schema validation failure for question {question_id}: "
        f"expected_elements from chunk has invalid type "
        f"{type(chunk_schema).__name__}, expected list, dict, or None "
        f"[correlation_id={correlation_id}]"
    )

# Return silently on dual-None without logging
if question_type == "none" and chunk_type == "none":
    return

# Enforce homogeneity allowing None compatibility
# None is compatible with any type, but non-None types must match
if question_type not in ("none", chunk_type) and chunk_type != "none":
    raise ValueError(
        f"Schema validation failure for question {question_id}: "
        f"heterogeneous types detected (question has {question_type}, "
        f"chunk has {chunk_type}) [correlation_id={correlation_id}]"
    )

# Validate list length equality
if question_type == "list" and chunk_type == "list":
    question_len = len(question_schema)
    chunk_len = len(chunk_schema)
    if question_len != chunk_len:
        raise ValueError(
            f"Schema validation failure for question {question_id}: "
            f"list length mismatch (question has {question_len} elements, "
            f"chunk has {chunk_len} elements) [correlation_id={correlation_id}]"
        )

# Validate dict key set equality with symmetric difference computation
```

```
    if question_type == "dict" and chunk_type == "dict":
        question_keys = set(question_schema.keys())
        chunk_keys = set(chunk_schema.keys())

        # Compute symmetric difference
        symmetric_diff = question_keys ^ chunk_keys

        if symmetric_diff:
            missing_in_chunk = question_keys - chunk_keys
            extra_in_chunk = chunk_keys - question_keys

            details = []
            if missing_in_chunk:
                details.append(f"missing in chunk: {sorted(missing_in_chunk)}")
            if extra_in_chunk:
                details.append(f"extra in chunk: {sorted(extra_in_chunk)}")

            raise ValueError(
                f"Schema validation failure for question {question_id}: "
                f"dict key set mismatch ({', '.join(details)}) "
                f"[correlation_id={correlation_id}]"
            )


def _validate_semantic_constraints(
    question_schema: Any,  # noqa: ANN401
    chunk_schema: Any,  # noqa: ANN401
    question_type: str,
    chunk_type: str,
    provisional_task_id: str,
    question_id: str,
    chunk_id: str,
    correlation_id: str,
) -> int:
    """Phase 6.3: Validate semantic constraints and return validated element count.

    Iterates deterministically via enumerate-zip for lists and sorted keys for
    dicts, extracts type-required-minimum fields with get defaults, implements
    asymmetric required implication as not-q-or-c boolean expression, enforces
    c-min-greater-equal-q-min threshold ordering, and returns validated element
    count.

    Args:
        question_schema: expected_elements from question
        chunk_schema: expected_elements from chunk
        question_type: Classified type of question schema
        chunk_type: Classified type of chunk schema
        provisional_task_id: Task ID for error reporting
        question_id: Question identifier for error messages
        chunk_id: Chunk identifier for error messages
        correlation_id: Correlation ID for distributed tracing

    Returns:
        Validated element count (number of elements validated)
```

```
    Raises:
            ValueError: If required field implication violated or threshold ordering
violated

    Semantic Constraints:
        - Asymmetric required implication: not q_required or c_required
        - Threshold ordering: c_minimum >= q_minimum
    """
    validated_count = 0

    # Iterate deterministically via enumerate-zip for lists
    if question_type == "list" and chunk_type == "list":
        for idx, (q_elem, c_elem) in enumerate(
            zip(question_schema, chunk_schema, strict=True)
        ):
            if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
                continue

            # Extract type-required-minimum fields with get defaults
            element_type = q_elem.get("type", f"element_at_index_{idx}")
            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)

            # Implement asymmetric required implication as not-q-or-c boolean expression
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field implication violation "
                    f"at index {idx}: element type '{element_type}' is required in "
                    f"question but marked as optional in chunk "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            # Enforce c-min-greater-equal-q-min threshold ordering
            if (
                isinstance(q_minimum, int | float)
                and isinstance(c_minimum, int | float)
                and c_minimum < q_minimum
            ):
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold ordering violation "
                    f"at index {idx}: element type '{element_type}' has chunk "
                    f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            validated_count += 1

    # Iterate deterministically via sorted keys for dicts
    elif question_type == "dict" and chunk_type == "dict":
```

```
        common_keys = set(question_schema.keys()) & set(chunk_schema.keys())

        for key in sorted(common_keys):
            q_elem = question_schema[key]
            c_elem = chunk_schema[key]

            if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
                continue

            # Extract type-required-minimum fields with get defaults
            element_type = q_elem.get("type", key)
            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)

            # Implement asymmetric required implication as not-q-or-c boolean expression
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field implication violation "
                    f"for key '{key}': element type '{element_type}' is required in "
                    f"question but marked as optional in chunk "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            # Enforce c-min-greater-equal-q-min threshold ordering
            if (
                isinstance(q_minimum, int | float)
                and isinstance(c_minimum, int | float)
                and c_minimum < q_minimum
            ):
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold ordering violation "
                    f"for key '{key}': element type '{element_type}' has chunk "
                    f"minimum ({c_minimum}) < question minimum ({q_minimum}) "
                    f"[question_id={question_id}, chunk_id={chunk_id}, "
                    f"correlation_id={correlation_id}]"
                )

            validated_count += 1

    return validated_count


def validate_phase6_schema_compatibility(
    question: dict[str, Any],
    chunk_expected_elements: list[dict[str, Any]] | dict[str, Any] | None,
    chunk_id: str,
    policy_area_id: str,
    correlation_id: str,
) -> int:
    """Phase 6.4: Orchestrator - Coordinate complete validation pipeline.
```

```
Invokes structural then semantic layers in sequence, captures element count
return value, emits debug log with has_required_fields and has_minimum_thresholds
computed via any-element-iteration, logs info warning for None chunk schema
with non-None question schema, and allows TypeError-ValueError propagation
to outer handler without try-except wrapping.

This is the main entry point for Phase 6 validation, designed to integrate
into the build_with_chunk_matrix loop after Phase 5 (signal resolution) and
before construct_task.

Args:
    question: Question dictionary from questionnaire
    chunk_expected_elements: expected_elements from chunk routing result
    chunk_id: Chunk identifier for logging
    policy_area_id: Policy area identifier for task ID construction
    correlation_id: Correlation ID for distributed tracing

Returns:
    Validated element count (number of elements validated)

Raises:
    TypeError: If either schema has invalid type (propagated from Phase 6.2)
    ValueError: If validation fails (propagated from Phase 6.1, 6.2, or 6.3)

Integration Point:
    Called within build_with_chunk_matrix loop:
    1. After Phase 5: Signal resolution completes
    2. Before construct_task: Task construction begins
    3. No try-except wrapper: Exceptions propagate to outer handler

Orchestration Flow:
    Phase 6.1: Extract and classify schemas
    Phase 6.2: Validate structural compatibility
    Phase 6.3: Validate semantic constraints (if both schemas non-None)
    Phase 6.4: Emit debug logs and return validated count
"""
question_id = question.get("question_id", "UNKNOWN")

# Phase 6.1: Classification & Extraction
(
    question_global,
    question_schema,
    chunk_schema,
    question_type,
    chunk_type,
) = _extract_and_classify_schemas(question, chunk_expected_elements, question_id)

# Construct provisional task ID for error reporting
provisional_task_id = f"MQC-{question_global:03d}_{policy_area_id}"

# Phase 6.2: Structural Validation
_validate_structural_compatibility(
    question_schema,
    chunk_schema,
```

```
    question_type,
    chunk_type,
    question_id,
    correlation_id,
)

# Phase 6.3: Semantic Validation (if both schemas non-None)
validated_count = 0
if question_schema is not None and chunk_schema is not None:
    validated_count = _validate_semantic_constraints(
        question_schema,
        chunk_schema,
        question_type,
        chunk_type,
        provisional_task_id,
        question_id,
        chunk_id,
        correlation_id,
    )

# Phase 6.4: Emit debug log with has_required_fields and has_minimum_thresholds
# Compute via any-element-iteration
has_required_fields = False
has_minimum_thresholds = False

if question_schema is not None:
    if isinstance(question_schema, list):
        has_required_fields = any(
            elem.get("required", False)
            for elem in question_schema
            if isinstance(elem, dict)
        )
        has_minimum_thresholds = any(
            "minimum" in elem for elem in question_schema if isinstance(elem, dict)
        )
    elif isinstance(question_schema, dict):
        has_required_fields = any(
            elem.get("required", False)
            for elem in question_schema.values()
            if isinstance(elem, dict)
        )
        has_minimum_thresholds = any(
            "minimum" in elem
            for elem in question_schema.values()
            if isinstance(elem, dict)
        )

logger.debug(
    f"Phase 6 validation complete: question_id={question_id}, "
    f"chunk_id={chunk_id}, provisional_task_id={provisional_task_id}, "
    f"validated_count={validated_count}, "
    f"has_required_fields={has_required_fields}, "
    f"has_minimum_thresholds={has_minimum_thresholds}, "
    f"question_type={question_type}, chunk_type={chunk_type}, "
```

```
        f"correlation_id={correlation_id}"
    )

    # Log info warning for None chunk schema with non-None question schema
    if question_schema is not None and chunk_schema is None:
        logger.info(
            f"Schema asymmetry detected: question_id={question_id}, "
            f"chunk_id={chunk_id}, question_schema_type={question_type}, "
            f"chunk_schema_type=none, message='Question specifies required elements "
            f"but chunk provides no schema', "
            f"validation_status='compatible_via_constraint_relaxation', "
            f"correlation_id={correlation_id}"
        )

    return validated_count


__all__ = [
    "validate_phase6_schema_compatibility",
    "_extract_and_classify_schemas",
    "_validate_structural_compatibility",
    "_validate_semantic_constraints",
    "_classify_expected_elements_type",
]
```

src/farfan_pipeline/phases/Phase_zero/__init__.py

```
"""
Phase Zero: Input Validation
============================

Phase 0 of the F.A.R.F.A.N pipeline responsible for validating and normalizing
input documents before they enter the main processing pipeline.

This phase ensures that all inputs meet the required preconditions for
downstream phases.
"""

__all__ = []
```

src/farfan_pipeline/phases/Phase_zero/boot_checks.py

```
"""
Boot-time validation checks for F.A.R.F.A.N runtime dependencies.

This module provides boot checks that validate critical dependencies before
pipeline execution. In PROD mode, failures abort execution unless explicitly
allowed by configuration flags.
"""

import importlib
import json
from pathlib import Path
from typing import Optional

from canonic_phases.Phase_zero.runtime_config import RuntimeConfig, RuntimeMode


class BootCheckError(Exception):
    """
    Raised when a boot check fails in strict mode.

    Attributes:
        component: Component that failed (e.g., "contradiction_module")
        reason: Human-readable failure reason
        code: Machine-readable error code
    """

    def __init__(self, component: str, reason: str, code: str):
        self.component = component
        self.reason = reason
        self.code = code
        super().__init__(f"Boot check failed [{code}] {component}: {reason}")


def check_contradiction_module_available(config: RuntimeConfig) -> bool:
    """
    Check if contradiction detection module is available.

    Args:
        config: Runtime configuration

    Returns:
        True if module available or fallback allowed

    Raises:
        BootCheckError: If module unavailable in strict PROD mode
    """
    try:
                          from   farfan_pipeline.analysis.contradiction_detection   import
PolicyContradictionDetector
        return True
    except ImportError as e:
        if config.mode == RuntimeMode.PROD and not config.allow_contradiction_fallback:
```

```
            raise BootCheckError(
                component="contradiction_module",
                reason=f"PolicyContradictionDetector not available: {e}",
                code="CONTRADICTION_MODULE_MISSING"
            )
        # Fallback allowed in DEV/EXPLORATORY or with flag
        return False


def check_wiring_validator_available(config: RuntimeConfig) -> bool:
    """
    Check if WiringValidator is available.

    Args:
        config: Runtime configuration

    Returns:
        True if validator available or disable allowed

    Raises:
        BootCheckError: If validator unavailable in strict PROD mode
    """
    try:
        from orchestration.wiring.validator import WiringValidator
        return True
    except ImportError as e:
        if config.mode == RuntimeMode.PROD and not config.allow_validator_disable:
            raise BootCheckError(
                component="wiring_validator",
                reason=f"WiringValidator not available: {e}",
                code="WIRING_VALIDATOR_MISSING"
            )
        return False


def check_spacy_model_available(model_name: str, config: RuntimeConfig) -> bool:
    """
    Check if preferred spaCy model is installed.

    Args:
        model_name: spaCy model name (e.g., "es_core_news_lg")
        config: Runtime configuration

    Returns:
        True if model available

    Raises:
        BootCheckError: If model unavailable in PROD mode
    """
    try:
        import spacy
        spacy.load(model_name)
        return True
    except (ImportError, OSError) as e:
```

```
        if config.mode == RuntimeMode.PROD:
            raise BootCheckError(
                component="spacy_model",
                reason=f"spaCy model '{model_name}' not available: {e}",
                code="SPACY_MODEL_MISSING"
            )
        return False


def check_calibration_files(config: RuntimeConfig, calibration_dir: Optional[Path] = None) -> bool:
    """
    Check calibration files exist and have required structure.

    Args:
        config: Runtime configuration
                  calibration_dir: Directory containing calibration files (default: config/layer_calibrations)

    Returns:
        True if calibration files valid

    Raises:
        BootCheckError: If calibration invalid in strict PROD mode
    """
    if calibration_dir is None:
        calibration_dir = Path("config/layer_calibrations")

    if not calibration_dir.exists():
        if config.mode == RuntimeMode.PROD and config.strict_calibration:
            raise BootCheckError(
                component="calibration_files",
                reason=f"Calibration directory not found: {calibration_dir}",
                code="CALIBRATION_DIR_MISSING"
            )
        return False

    # Check for required calibration files
    required_files = ["intrinsic_calibration.json", "fusion_specification.json"]
    missing_files = []

    for filename in required_files:
        file_path = calibration_dir.parent / filename
        if not file_path.exists():
            missing_files.append(filename)

    if missing_files:
        if config.mode == RuntimeMode.PROD and config.strict_calibration:
            raise BootCheckError(
                component="calibration_files",
                                reason=f"Missing required calibration files: {', '.join(missing_files)}",
                code="CALIBRATION_FILES_MISSING"
            )
```

```
        return False

    # Validate _base_weights presence in strict mode
    if config.strict_calibration:
        intrinsic_path = calibration_dir.parent / "intrinsic_calibration.json"
        try:
            with open(intrinsic_path) as f:
                data = json.load(f)

            if "_base_weights" not in data:
                if config.mode == RuntimeMode.PROD:
                    raise BootCheckError(
                        component="calibration_files",
                        reason=f"Missing _base_weights in {intrinsic_path}",
                        code="CALIBRATION_BASE_WEIGHTS_MISSING"
                    )
                return False
        except (json.JSONDecodeError, IOError) as e:
            if config.mode == RuntimeMode.PROD:
                raise BootCheckError(
                    component="calibration_files",
                    reason=f"Failed to parse {intrinsic_path}: {e}",
                    code="CALIBRATION_PARSE_ERROR"
                )
            return False

    return True


def check_orchestration_metrics_contract(config: RuntimeConfig) -> bool:
    """
    Check that orchestration metrics contract is properly defined.

    This validates that the _execution_metrics['phase_2'] schema exists
    and is properly structured.

    Args:
        config: Runtime configuration

    Returns:
        True if contract valid

    Raises:
        BootCheckError: If contract invalid in PROD mode
    """
    try:
        # Import orchestrator to check metrics contract
        from orchestration.orchestrator import Orchestrator

        # Verify phase_2 metrics schema exists
        # This is a placeholder - actual implementation would check the schema
        return True
    except ImportError as e:
        if config.mode == RuntimeMode.PROD:
```

```
            raise BootCheckError(
                component="orchestration_metrics",
                reason=f"Orchestrator not available: {e}",
                code="ORCHESTRATOR_MISSING"
            )
        return False


def check_networkx_available() -> bool:
    """
    Check if NetworkX is available for graph metrics.

    Returns:
        True if NetworkX available

    Note:
             This is a soft check - NetworkX unavailability is Category B (quality
degradation)
    """
    try:
        import networkx
        return True
    except ImportError:
        return False


def run_boot_checks(config: RuntimeConfig) -> dict[str, bool]:
    """
    Run all boot checks and return results.

    Args:
        config: Runtime configuration

    Returns:
        Dictionary mapping check name to success status

    Raises:
        BootCheckError: If any critical check fails in strict PROD mode

    Example:
        >>> config = RuntimeConfig.from_env()
        >>> results = run_boot_checks(config)
        >>> assert results['contradiction_module']
    """
    results = {}

    # Critical checks (Category A)
    results['contradiction_module'] = check_contradiction_module_available(config)
    results['wiring_validator'] = check_wiring_validator_available(config)
      results['spacy_model'] = check_spacy_model_available(config.preferred_spacy_model,
config)
    results['calibration_files'] = check_calibration_files(config)
    results['orchestration_metrics'] = check_orchestration_metrics_contract(config)
```

```
    # Quality checks (Category B)
    results['networkx'] = check_networkx_available()

    return results


def get_boot_check_summary(results: dict[str, bool]) -> str:
    """
    Generate human-readable summary of boot check results.

    Args:
        results: Boot check results from run_boot_checks()

    Returns:
        Formatted summary string
    """
    passed = sum(1 for v in results.values() if v)
    total = len(results)

    lines = [f"Boot Checks: {passed}/{total} passed"]
    lines.append("")

    for check, success in results.items():
        status = "?" if success else "?"
        lines.append(f"  {status} {check}")

    return "\n".join(lines)
```

src/farfan_pipeline/phases/Phase_zero/bootstrap.py

```
"""Bootstrap module for deterministic wiring initialization.

Implements the complete initialization sequence with:
1. Resource loading (QuestionnaireResourceProvider)
2. Signal system setup (memory:// by default, HTTP optional)
3. CoreModuleFactory with DI
4. ArgRouterExtended (?30 routes)
5. Orchestrator assembly

All initialization is deterministic and observable.
"""

from __future__ import annotations

import json
import time
from collections import OrderedDict
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any

import structlog

from canonic_phases.Phase_zero.paths import CONFIG_DIR, DATA_DIR
from orchestration.factory import CanonicalQuestionnaire
from canonic_phases.Phase_two.arg_router import ExtendedArgRouter
from orchestration.class_registry import build_class_registry
from canonic_phases.Phase_two.executor_config import ExecutorConfig
from orchestration.factory import CoreModuleFactory
from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signals import (
    InMemorySignalSource,
    SignalClient,
    SignalPack,
    SignalRegistry,
)
from      cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_consumption
import (
    AccessLevel,
    get_access_audit,
)

@dataclass
class QuestionnaireResourceProvider:
    """
    Proveedor de recursos del cuestionario con acceso scoped.

    NIVEL 2: Acceso Parcial Recurrente
    PROVEEDOR: AnalysisPipelineFactory (Nivel 1)
    CONSUMIDORES: Orchestrator, ReportAssembler, SISAS components

    Este provider NO hace I/O. Recibe CanonicalQuestionnaire ya cargado
    y expone métodos que retornan subconjuntos específicos.
```

```
"""
questionnaire_path: Path | None = None
data_dir: Path = field(default_factory=lambda: DATA_DIR)
_canonical: CanonicalQuestionnaire | None = field(default=None, repr=False)

# === MÉTODOS DE INICIALIZACIÓN ===

def initialize(self, canonical: CanonicalQuestionnaire) -> None:
    """
    Inicializa el provider con el cuestionario canónico.
    DEBE ser llamado por Factory después de load_questionnaire().
    """
    object.__setattr__(self, '_canonical', canonical)

@property
def is_initialized(self) -> bool:
    """Verifica si el provider fue inicializado."""
    return self._canonical is not None

def _require_initialized(self) -> CanonicalQuestionnaire:
    """Guarda que lanza error si no está inicializado."""
    if self._canonical is None:
        raise RuntimeError(
            "QuestionnaireResourceProvider not initialized. "
            "Call initialize() with CanonicalQuestionnaire first."
        )
    return self._canonical

# === MÉTODOS DE ACCESO SCOPED (NIVEL 2) ===

def get_data(self) -> dict[str, Any]:
    """
    Retorna el contenido completo del cuestionario.
    SCOPE: Total (pero solo lectura, no I/O)
    USO: ReportAssembler para metadatos
    """
    return dict(self._require_initialized().data)

def get_dimensions(self) -> dict[str, Any]:
    """
    SCOPE: Solo canonical_notation.dimensions (6 items)
    """
    result = self._require_initialized().dimensions
    get_access_audit().record_access(
        level=AccessLevel.ORCHESTRATOR,
        accessor_module=__name__,
        accessor_class="QuestionnaireResourceProvider",
        accessor_method="get_dimensions",
        accessed_block="dimensions",
        accessed_keys=list(result.keys()),
    )
    return result

def get_policy_areas(self) -> dict[str, Any]:
```

```
        """
        SCOPE: Solo canonical_notation.policy_areas (10 items)
        """
        result = self._require_initialized().policy_areas
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_policy_areas",
            accessed_block="policy_areas",
            accessed_keys=list(result.keys()),
        )
        return result

    def get_micro_questions_for_policy_area(self, pa_id: str) -> list[dict[str, Any]]:
        """
        SCOPE: Micro preguntas filtradas por policy_area_id
        Ejemplo: get_micro_questions_for_policy_area("PA01") ? 30 preguntas
        """
        canonical = self._require_initialized()
          result = [q for q in canonical.micro_questions if q.get("policy_area_id") ==
pa_id]
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_micro_questions_for_policy_area",
            accessed_block="micro_questions",
            accessed_keys=[q.get("question_id", "") for q in result],
            scope_filter=f"policy_area_id={pa_id}",
        )
        return result

    def get_micro_questions_for_dimension(self, dim_id: str) -> list[dict[str, Any]]:
        """
        SCOPE: Micro preguntas filtradas por dimension_id
        Ejemplo: get_micro_questions_for_dimension("DIM01") ? 50 preguntas
        """
        canonical = self._require_initialized()
           result = [q for q in canonical.micro_questions if q.get("dimension_id") ==
dim_id]
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_micro_questions_for_dimension",
            accessed_block="micro_questions",
            accessed_keys=[q.get("question_id", "") for q in result],
            scope_filter=f"dimension_id={dim_id}",
        )
        return result

    def get_patterns_by_question(self, question_id: str) -> list[dict[str, Any]]:
        """
```

```
        SCOPE: Patrones de una micro pregunta específica
        USO: ReportAssembler para enrichment de respuestas
        """
        canonical = self._require_initialized()
        result = []
        for q in canonical.micro_questions:
            if q.get("question_id") == question_id:
                result = list(q.get("patterns", []))
                break
        pattern_ids = [p.get("id", f"idx_{i}") for i, p in enumerate(result)]
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_patterns_by_question",
            accessed_block="patterns",
            accessed_keys=pattern_ids,
            scope_filter=f"question_id={question_id}",
        )
        return result

    def get_expected_elements_for_question(self, question_id: str) -> list[dict[str,
Any]]:
        """
        SCOPE: Elementos esperados de una micro pregunta específica
        USO: EvidenceValidator para verificación
        """
        canonical = self._require_initialized()
        result = []
        for q in canonical.micro_questions:
            if q.get("question_id") == question_id:
                result = list(q.get("expected_elements", []))
                break
        element_types = [e.get("type", f"idx_{i}") for i, e in enumerate(result)]
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_expected_elements_for_question",
            accessed_block="expected_elements",
            accessed_keys=element_types,
            scope_filter=f"question_id={question_id}",
        )
        return result

    def get_failure_contract_for_question(self, question_id: str) -> dict[str, Any] |
None:
        """
        SCOPE: Contrato de falla de una micro pregunta específica
        USO: EvidenceValidator para abort conditions
        """
        canonical = self._require_initialized()
        result = None
        for q in canonical.micro_questions:
```

```
            if q.get("question_id") == question_id:
                result = q.get("failure_contract")
                break
        get_access_audit().record_access(
            level=AccessLevel.ORCHESTRATOR,
            accessor_module=__name__,
            accessor_class="QuestionnaireResourceProvider",
            accessor_method="get_failure_contract_for_question",
            accessed_block="failure_contract",
            accessed_keys=[question_id] if result else [],
            scope_filter=f"question_id={question_id}",
        )
        return result

    @property
    def source_hash(self) -> str:
        """Hash SHA256 del cuestionario para trazabilidad."""
        if self._canonical is None:
            return ""
        return self._canonical.sha256

_CalibrationOrchestrator = None  # type: ignore[assignment]
_DEFAULT_CALIBRATION_CONFIG = None  # type: ignore[assignment]
_HAS_CALIBRATION = False

from          orchestration.wiring.errors          import          MissingDependencyError,
WiringInitializationError
from orchestration.wiring.feature_flags import WiringFeatureFlags
from orchestration.wiring.phase_0_validator import Phase0Validator
from orchestration.wiring.validation import WiringValidator

logger = structlog.get_logger(__name__)


@dataclass
class WiringComponents:
    """Container for all wired components.

    Attributes:
        provider: QuestionnaireResourceProvider
        signal_client: SignalClient (memory:// or HTTP)
        signal_registry: SignalRegistry with TTL and LRU
        executor_config: ExecutorConfig with defaults
        factory: CoreModuleFactory with DI
        arg_router: ExtendedArgRouter with special routes
        class_registry: Class registry for routing
        validator: WiringValidator for contract checking
        flags: Feature flags used during initialization
        init_hashes: Hashes computed during initialization
    """

    provider: QuestionnaireResourceProvider
    signal_client: SignalClient
    signal_registry: SignalRegistry
```

```
    executor_config: ExecutorConfig
    factory: CoreModuleFactory
    arg_router: ExtendedArgRouter
    class_registry: dict[str, type]
    validator: WiringValidator
    flags: WiringFeatureFlags
    calibration_orchestrator: "_CalibrationOrchestrator | None" = None
    init_hashes: dict[str, str] = field(default_factory=dict)


CANONICAL_POLICY_AREA_DEFINITIONS: "OrderedDict[str, dict[str, list[str] | str]]" =
OrderedDict(
    [
        (
            "PA01",
            {
                "name": "Derechos de las mujeres e igualdad de género",
                "slug": "genero_mujeres",
                "aliases": ["fiscal"],
            },
        ),
        (
            "PA02",
            {
                "name": "Prevención de la violencia y protección",
                "slug": "seguridad_violencia",
                "aliases": ["salud"],
            },
        ),
        (
            "PA03",
            {
                "name": "Ambiente sano y cambio climático",
                "slug": "ambiente",
                "aliases": ["ambiente"],
            },
        ),
        (
            "PA04",
            {
                "name": "Derechos económicos, sociales y culturales",
                "slug": "derechos_sociales",
                "aliases": ["energía"],
            },
        ),
        (
            "PA05",
            {
                "name": "Derechos de las víctimas y construcción de paz",
                "slug": "paz_victimas",
                "aliases": ["transporte"],
            },
        ),
        (
```

```
            "PA06",
            {
                "name": "Derecho al futuro de la niñez y juventud",
                "slug": "ninez_juventud",
                "aliases": [],
            },
        ),
        (
            "PA07",
            {
                "name": "Tierras y territorios",
                "slug": "tierras_territorios",
                "aliases": [],
            },
        ),
        (
            "PA08",
            {
                "name": "Líderes, lideresas y defensores de DD. HH.",
                "slug": "liderazgos_ddhh",
                "aliases": [],
            },
        ),
        (
            "PA09",
            {
                "name": "Derechos de personas privadas de libertad",
                "slug": "privados_libertad",
                "aliases": [],
            },
        ),
        (
            "PA10",
            {
                "name": "Migración transfronteriza",
                "slug": "migracion",
                "aliases": [],
            },
        ),
    ]
)

SIGNAL_PACK_VERSION = "1.0.0"
MAX_PATTERNS_PER_POLICY_AREA = 32

class WiringBootstrap:
    """Bootstrap engine for deterministic wiring initialization.

    Follows strict initialization order:
    1. Load resources (questionnaire)
    2. Build signal system (memory:// or HTTP)
    3. Create factory with DI
    4. Initialize arg router
    5. Validate all contracts
```

```
    """

    def __init__(
        self,
        questionnaire_path: str | Path,
        questionnaire_hash: str,
        executor_config_path: str | Path,
        calibration_profile: str,
        abort_on_insufficient: bool,
        resource_limits: dict[str, int],
        flags: WiringFeatureFlags | None = None,
    ) -> None:
        """Initialize bootstrap engine.

        Args:
            questionnaire_path: Path to questionnaire monolith JSON.
            questionnaire_hash: Expected SHA-256 hash of the monolith.
            executor_config_path: Path to the executor configuration.
            calibration_profile: The calibration profile to use.
            abort_on_insufficient: Flag to abort on insufficient data.
            resource_limits: Resource limit settings.
            flags: Feature flags (defaults to environment).
        """
        self.questionnaire_path = questionnaire_path
        self.questionnaire_hash = questionnaire_hash
        self.executor_config_path = executor_config_path
        self.calibration_profile = calibration_profile
        self.abort_on_insufficient = abort_on_insufficient
        self.resource_limits = resource_limits
        self.flags = flags or WiringFeatureFlags.from_env()
        self._start_time = time.time()

        # Validate flags
        warnings = self.flags.validate()
        for warning in warnings:
            logger.warning("feature_flag_warning", message=warning)

        logger.info(
            "wiring_bootstrap_initialized",
            questionnaire_path=str(questionnaire_path) if questionnaire_path else None,
            flags=self.flags.to_dict(),
        )

    def bootstrap(self) -> WiringComponents:
        """Execute complete bootstrap sequence.

        Returns:
            WiringComponents with all initialized modules

        Raises:
            WiringInitializationError: If any phase fails
        """
        logger.info("wiring_bootstrap_start")
```

```
try:
    # Phase 0: Validate configuration contract
    logger.info("wiring_init_phase", phase="phase_0_validation")
    phase_0_validator = Phase0Validator()
    raw_config = {
        "monolith_path": self.questionnaire_path,
        "questionnaire_hash": self.questionnaire_hash,
        "executor_config_path": self.executor_config_path,
        "calibration_profile": self.calibration_profile,
        "abort_on_insufficient": self.abort_on_insufficient,
        "resource_limits": self.resource_limits,
    }
    phase_0_validator.validate(raw_config)
    logger.info("phase_0_validation_passed")

    # Phase 1: Load resources
    provider = self._load_resources()

    # Phase 2: Build signal system
    signal_client, signal_registry = self._build_signal_system(provider)

    # Phase 3: Create executor config
    executor_config = self._create_executor_config()

    # Phase 4: Create factory with DI
    factory = self._create_factory(provider, signal_registry, executor_config)

    # Phase 5: Build class registry
    class_registry = self._build_class_registry()

    # Phase 6: Initialize arg router
    arg_router = self._create_arg_router(class_registry)

    # Phase 7: Create validator
    validator = WiringValidator()

    # Phase 8: Create calibration orchestrator (optional enhancement)
    calibration_orchestrator = self._create_calibration_orchestrator()

    # Phase 9: Seed signals (if memory mode)
    if signal_client._transport == "memory":
        metrics = self._seed_canonical_policy_area_signals(
            signal_client._memory_source,
            signal_registry,
            provider,
        )
        logger.info(
            "signals_seeded",
            areas=metrics["canonical_areas"],
            aliases=metrics["legacy_aliases"],
            hit_rate=metrics["hit_rate"],
        )

    # Compute initialization hashes
```

```
        init_hashes = self._compute_init_hashes(
            provider, signal_registry, factory, arg_router
        )

        components = WiringComponents(
            provider=provider,
            signal_client=signal_client,
            signal_registry=signal_registry,
            executor_config=executor_config,
            factory=factory,
            arg_router=arg_router,
            class_registry=class_registry,
            validator=validator,
            calibration_orchestrator=calibration_orchestrator,
            flags=self.flags,
            init_hashes=init_hashes,
        )

        elapsed = time.time() - self._start_time

        logger.info(
            "wiring_bootstrap_complete",
            elapsed_s=elapsed,
            factory_instances=19,  # Expected count
            argrouter_routes=arg_router.get_special_route_coverage(),
            signals_mode=signal_client._transport,
            init_hashes={k: v[:16] for k, v in init_hashes.items()},
        )

        return components

    except Exception as e:
        elapsed = time.time() - self._start_time
        logger.error(
            "wiring_bootstrap_failed",
            elapsed_s=elapsed,
            error=str(e),
            error_type=type(e).__name__,
        )
        raise

def _load_resources(self) -> QuestionnaireResourceProvider:
    """Load questionnaire resources.

    Returns:
        QuestionnaireResourceProvider instance

    Raises:
        WiringInitializationError: If loading fails
    """
    logger.info("wiring_init_phase", phase="load_resources")

    try:
        if self.questionnaire_path:
```

```
                path = Path(self.questionnaire_path)
                if not path.exists():
                    raise MissingDependencyError(
                        dependency=str(path),
                        required_by="WiringBootstrap",
                        fix=f"Ensure questionnaire file exists at {path}",
                    )

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

                provider = QuestionnaireResourceProvider(data)
            else:
                # Use default/empty provider
                provider = QuestionnaireResourceProvider({})

            logger.info(
                "questionnaire_loaded",
                       path=str(self.questionnaire_path) if self.questionnaire_path else
"default",
            )

            return provider

        except Exception as e:
            raise WiringInitializationError(
                phase="load_resources",
                component="QuestionnaireResourceProvider",
                reason=str(e),
            ) from e

    def _build_signal_system(
        self,
        provider: QuestionnaireResourceProvider,
    ) -> tuple[SignalClient, SignalRegistry]:
        """Build signal system (memory:// or HTTP).

        Args:
            provider: QuestionnaireResourceProvider for signal data

        Returns:
            Tuple of (SignalClient, SignalRegistry)

        Raises:
            WiringInitializationError: If setup fails
        """
        logger.info("wiring_init_phase", phase="build_signal_system")

        try:
            # Create registry first
            registry = SignalRegistry(
                max_size=100,
                default_ttl_s=3600,
            )
```

```
        # Create signal source
        if self.flags.enable_http_signals:
            # HTTP mode (requires explicit configuration)
            base_url = "http://127.0.0.1:8000"  # Default, should be configurable
            logger.info("signal_client_http_mode", base_url=base_url)

            client = SignalClient(
                base_url=base_url,
                enable_http_signals=True,
            )
        else:
            # Memory mode (default)
            memory_source = InMemorySignalSource()

            client = SignalClient(
                base_url="memory://",
                enable_http_signals=False,
                memory_source=memory_source,
            )

            logger.info("signal_client_memory_mode")

        return client, registry

    except Exception as e:
        raise WiringInitializationError(
            phase="build_signal_system",
            component="SignalClient/SignalRegistry",
            reason=str(e),
        ) from e

def _create_executor_config(self) -> ExecutorConfig:
    """Create executor configuration.

    Returns:
        ExecutorConfig with defaults
    """
    logger.info("wiring_init_phase", phase="create_executor_config")

    config = ExecutorConfig(
        max_tokens=2048,
        temperature=0.0,  # Deterministic
        timeout_s=30.0,
        retry=2,
        seed=0 if self.flags.deterministic_mode else None,
    )

    logger.info(
        "executor_config_created",
        deterministic=self.flags.deterministic_mode,
        seed=config.seed,
    )
```

```
    return config

def _create_factory(
    self,
    provider: QuestionnaireResourceProvider,
    registry: SignalRegistry,
    config: ExecutorConfig,
) -> CoreModuleFactory:
    """Create CoreModuleFactory with DI.

    Args:
        provider: QuestionnaireResourceProvider
        registry: SignalRegistry for injection
        config: ExecutorConfig for injection

    Returns:
        CoreModuleFactory instance

    Raises:
        WiringInitializationError: If creation fails
    """
    logger.info("wiring_init_phase", phase="create_factory")

    try:
        factory = CoreModuleFactory(
            data_dir=provider.data_dir,
        )

        logger.info(
            "factory_created",
            data_dir=str(provider.data_dir),
        )

        return factory

    except Exception as e:
        raise WiringInitializationError(
            phase="create_factory",
            component="CoreModuleFactory",
            reason=str(e),
        ) from e

def _build_class_registry(self) -> dict[str, type]:
    """Build class registry for arg router.

    Returns:
        Class registry mapping names to types

    Raises:
        WiringInitializationError: If build fails
    """
    logger.info("wiring_init_phase", phase="build_class_registry")

    try:
```

```
        registry = build_class_registry()

        logger.info(
            "class_registry_built",
            class_count=len(registry),
        )

        return registry

    except Exception as e:
        raise WiringInitializationError(
            phase="build_class_registry",
            component="ClassRegistry",
            reason=str(e),
        ) from e

def _create_arg_router(
    self,
    class_registry: dict[str, type],
) -> ExtendedArgRouter:
    """Create ExtendedArgRouter with special routes.

    Args:
        class_registry: Class registry for routing

    Returns:
        ExtendedArgRouter instance

    Raises:
        WiringInitializationError: If creation fails
    """
    logger.info("wiring_init_phase", phase="create_arg_router")

    try:
        router = ExtendedArgRouter(class_registry)

        route_count = router.get_special_route_coverage()

        if route_count < 30:
            logger.warning(
                "argrouter_coverage_low",
                count=route_count,
                expected=30,
            )

        logger.info(
            "arg_router_created",
            special_routes=route_count,
        )

        return router

    except Exception as e:
        raise WiringInitializationError(
```

```
                phase="create_arg_router",
                component="ExtendedArgRouter",
                reason=str(e),
            ) from e

    def _create_calibration_orchestrator(self) -> "_CalibrationOrchestrator | None":
        """
        Create CalibrationOrchestrator when calibration stack is available.

        Returns:
            CalibrationOrchestrator instance or None if unavailable.
        """
                if not _HAS_CALIBRATION or _CalibrationOrchestrator is None or
_DEFAULT_CALIBRATION_CONFIG is None:
            logger.info("calibration_system_unavailable")
            return None

        data_dir = DATA_DIR
        config_dir = CONFIG_DIR

        kwargs: dict[str, Any] = {"config": _DEFAULT_CALIBRATION_CONFIG}

        intrinsic_path = config_dir / "intrinsic_calibration.json"
        if intrinsic_path.exists():
            kwargs["intrinsic_calibration_path"] = intrinsic_path

        compatibility_path = data_dir / "method_compatibility.json"
        if compatibility_path.exists():
            kwargs["compatibility_path"] = compatibility_path

        registry_path = data_dir / "method_registry.json"
        if registry_path.exists():
            kwargs["method_registry_path"] = registry_path

        signatures_path = data_dir / "method_signatures.json"
        if signatures_path.exists():
            kwargs["method_signatures_path"] = signatures_path

        try:
            orchestrator = _CalibrationOrchestrator(**kwargs)
            logger.info(
                "calibration_orchestrator_ready",
                intrinsic=str(intrinsic_path),
                compatibility=str(compatibility_path),
            )
            return orchestrator
        except Exception as exc:  # pragma: no cover - defensive guardrail
            logger.warning(
                "calibration_orchestrator_initialization_failed",
                error=str(exc),
            )
            return None

    def _build_signal_pack(
```

```
        self,
        provider: QuestionnaireResourceProvider,
        canonical_id: str,
        meta: dict[str, Any],
        *,
        alias: str | None = None,
    ) -> SignalPack:
        """Build a SignalPack for a canonical policy area (and optional alias)."""
        pattern_source = getattr(provider, "get_patterns_for_area", None)
             patterns = pattern_source(canonical_id, MAX_PATTERNS_PER_POLICY_AREA) if
callable(pattern_source) else []

        pack = SignalPack(
            version=SIGNAL_PACK_VERSION,
            policy_area=alias or canonical_id,  # type: ignore[arg-type]
            patterns=patterns,
            metadata={
                "canonical_id": canonical_id,
                "display_name": meta["name"],
                "slug": meta["slug"],
                "alias": alias,
            },
        )
        fingerprint = pack.compute_hash()
        return pack.model_copy(update={"source_fingerprint": fingerprint})

    @staticmethod
    def _register_signal_pack(
        memory_source: InMemorySignalSource,
        registry: SignalRegistry,
        pack: SignalPack,
    ) -> None:
        """Register pack in both memory source and registry."""
        memory_source.register(pack.policy_area, pack)
        registry.put(pack.policy_area, pack)
        logger.debug(
            "signal_seeded",
            policy_area=pack.policy_area,
            canonical_id=pack.metadata.get("canonical_id"),
            patterns=len(pack.patterns),
        )

    def _seed_canonical_policy_area_signals(
        self,
        memory_source: InMemorySignalSource,
        registry: SignalRegistry,
        provider: QuestionnaireResourceProvider,
    ) -> dict[str, Any]:
        """
        Seed signal registry with canonical (PA01-PA10) policy areas.

        Returns:
            Metrics dict with coverage and legacy alias info.
        """
```

```
    canonical_count = 0
    alias_count = 0

    for area_id, meta in CANONICAL_POLICY_AREA_DEFINITIONS.items():
        pack = self._build_signal_pack(provider, area_id, meta)
        self._register_signal_pack(memory_source, registry, pack)
        canonical_count += 1

        for alias in meta["aliases"]:  # type: ignore[index]
            alias_pack = self._build_signal_pack(
                provider,
                area_id,
                meta,
                alias=alias,
            )
            self._register_signal_pack(memory_source, registry, alias_pack)
            alias_count += 1

    hits = sum(
        1
        for area_id in CANONICAL_POLICY_AREA_DEFINITIONS
        if registry.get(area_id) is not None
    )
    total_required = len(CANONICAL_POLICY_AREA_DEFINITIONS)
    hit_rate = hits / total_required if total_required else 0.0

    return {
        "canonical_areas": canonical_count,
        "legacy_aliases": alias_count,
        "hit_rate": hit_rate,
        "required_hit_rate": 0.95,
    }

def seed_signals_public(
    self,
    client: SignalClient,
    registry: SignalRegistry,
    provider: QuestionnaireResourceProvider,
) -> dict[str, Any]:
    """Seed initial signals in memory mode (PUBLIC API).

    This replaces the private _seed_signals method with a public API that:
    1. Validates the SignalClient is using memory transport
    2. Returns deterministic metrics for validation
    3. Enforces the ?95% hit rate requirement

    Args:
        client: SignalClient to seed (must be in memory mode)
        registry: SignalRegistry to populate
        provider: QuestionnaireResourceProvider for patterns

    Returns:
        Dict with seeding metrics (areas_seeded, total_signals, hit_rate)
```

```
    Raises:
        ValueError: If client is not in memory mode
        WiringInitializationError: If hit rate requirement is not met
    """
    logger.info("wiring_init_phase", phase="seed_signals_public")

    if getattr(client, "_transport", None) != "memory":
        raise ValueError(
            "Signal seeding requires memory mode. "
            "Set enable_http_signals=False in WiringFeatureFlags."
        )

    memory_source = getattr(client, "_memory_source", None)
    if memory_source is None:
        raise ValueError("Signal client memory source not initialized.")

    metrics = self._seed_canonical_policy_area_signals(
        memory_source,
        registry,
        provider,
    )

    if metrics["hit_rate"] < metrics["required_hit_rate"]:
        raise WiringInitializationError(
            phase="seed_signals",
            component="SignalRegistry",
            reason=(
                f"Signal hit rate {metrics['hit_rate']:.2%} below "
                f"required threshold {metrics['required_hit_rate']:.2%}"
            ),
        )

    return metrics



def _compute_init_hashes(
    self,
    provider: QuestionnaireResourceProvider,
    registry: SignalRegistry,
    factory: CoreModuleFactory,
    router: ExtendedArgRouter,
) -> dict[str, str]:
    """Compute hashes for initialized components.

    Args:
        provider: QuestionnaireResourceProvider
        registry: SignalRegistry
        factory: CoreModuleFactory
        router: ExtendedArgRouter

    Returns:
        Dict of component names to their hashes
    """
```

```
        import blake3

        hashes = {}

        # Provider hash (based on data keys)
         provider_keys = sorted(provider._data.keys()) if hasattr(provider, '_data') else
[]
        hashes["provider"] = blake3.blake3(
            json.dumps(provider_keys, sort_keys=True).encode('utf-8')
        ).hexdigest()

        # Registry hash (based on metrics)
        registry_metrics = registry.get_metrics()
        hashes["registry"] = blake3.blake3(
            json.dumps(registry_metrics, sort_keys=True).encode('utf-8')
        ).hexdigest()

        # Router hash (based on special routes count)
        router_data = {"route_count": router.get_special_route_coverage()}
        hashes["router"] = blake3.blake3(
            json.dumps(router_data, sort_keys=True).encode('utf-8')
        ).hexdigest()

        return hashes


__all__ = [
    'WiringComponents',
    'WiringBootstrap',
]
```