Phase 2 Source Code

```
================================================================================
PHASE 2: COMPLETE SOURCE CODE COMPILATION
Generated: 2025-12-20T21:07:55.915544Z
Total files: 29
================================================================================

TABLE OF CONTENTS
----------------------------------------
```

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


################################################################################
# FILE: __init__.py
# PATH: src/farfan_pipeline/phases/Phase_two/__init__.py
################################################################################

"""
Module: Phase_two/__init__
PHASE_LABEL: Phase 2
Sequence: Package Init
Description: Phase 2 package interface and exports

Version: 1.0.0
Last Modified: 2025-12-20
Author: F.A.R.F.A.N Policy Pipeline
License: Proprietary

Phase 2: Analysis & Question Execution - Contract-Driven Processing.

This phase implements contract-driven question execution with evidence assembly,
narrative synthesis, and SISAS integration for deterministic policy analysis.

File Sequence (a-z, aa-ac):
- phase2_a_arg_router.py          : Argument routing
- phase2_b_base_executor_with_contract.py : Executor base class
- phase2_c_carver.py              : Narrative synthesis
- phase2_d_calibration_policy.py  : Calibration policies
- phase2_e_contract_validator_cqvr.py : Contract validation
- phase2_f_evidence_nexus.py      : Evidence assembly
- phase2_g_method_signature_validator.py : Signature validation
- phase2_h_metrics_persistence.py : Metrics persistence
- phase2_i_precision_tracking.py  : Precision tracking
- phase2_j_resource_integration.py : Resource integration
- phase2_k_resource_aware_executor.py : Resource-aware executor
- phase2_l_resource_manager.py    : Resource management
- phase2_m_signature_runtime_validator.py : Runtime validation
- phase2_n_task_planner.py        : Task planning
- phase2_o_methods_registry.py    : Methods registry
- phase2_p_executor_profiler.py   : Profiling
- phase2_q_executor_instrumentation_mixin.py : Instrumentation
- phase2_r_executor_calibration_integration.py : Calibration integration
- phase2_s_executor_config.py     : Executor config
- phase2_t_irrigation_synchronizer.py : Signal irrigation
- phase2_u_synchronization.py     : Sync utilities
- phase2_v_executor_chunk_synchronizer.py : Chunk sync
- phase2_w_factory.py             : DI Factory
- phase2_x_class_registry.py      : Class registry
- phase2_y_schema_validation.py   : Schema validation
- phase2_z_generic_contract_executor.py : Generic executor
- phase2_aa_method_source_validator.py : Source validation
- phase2_ab_resource_alerts.py    : Resource alerts
- phase2_ac_executor_tests.py     : Executor tests
"""

from __future__ import annotations
```

Phase 2 Source Code

```python
from typing import TYPE_CHECKING

# Evidence processing - EvidenceNexus for causal graph construction
from farfan_pipeline.phases.Phase_two.phase2_f_evidence_nexus import (
    EvidenceNexus,
    EvidenceGraph,
    EvidenceNode,
    process_evidence,
)

# Narrative synthesis - Doctoral Carver for PhD-level responses
from farfan_pipeline.phases.Phase_two.phase2_c_carver import (
    DoctoralCarverSynthesizer,
    CarverAnswer,
)

# Executor configuration and base class
from farfan_pipeline.phases.Phase_two.phase2_s_executor_config import ExecutorConfig
from farfan_pipeline.phases.Phase_two.phase2_b_base_executor_with_contract import (
    BaseExecutorWithContract,
)

__all__ = [
    # Evidence processing (EvidenceNexus)
    "EvidenceNexus",
    "EvidenceGraph",
    "EvidenceNode",
    "process_evidence",
    # Narrative synthesis (Carver)
    "DoctoralCarverSynthesizer",
    "CarverAnswer",
    # Executor configuration
    "ExecutorConfig",
    "BaseExecutorWithContract",
]



################################################################################
# FILE: phase2_a_arg_router.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_a_arg_router.py
################################################################################

"""Argument routing with special routes, strict validation, and comprehensive metrics.

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Argument Router
PHASE_ROLE: Routes arguments to executor methods with special handlers and strict validation

This module provides ExtendedArgRouter (and legacy ArgRouter for compatibility):
- 30+ special route handlers for commonly-called methods
- Strict validation (no silent parameter drops)
- **kwargs support for forward compatibility
- Full observability and metrics
- Base routing and validation utilities

Design Principles:
- Explicit route definitions for high-traffic methods
- Fail-fast on missing required arguments
- Fail-fast on unexpected arguments (unless **kwargs present)
- Full traceability of routing decisions
- Zero tolerance for silent parameter drops
"""

from __future__ import annotations

import inspect
import logging
import os
import random
import threading
from collections.abc import Iterable, Mapping, MutableMapping
from dataclasses import dataclass
from typing import (
    Any,
    Union,
    get_args,
    get_origin,
    get_type_hints,
)

try:
    import structlog  # type: ignore
except Exception:  # pragma: no cover
    structlog = None

std_logger = logging.getLogger(__name__)

if structlog is None:
    class _CompatLogger:
        def __init__(self, base: logging.Logger) -> None:
            self._base = base

        def debug(self, event: str, **kwargs: Any) -> None:
            self._base.debug("%s %s", event, kwargs if kwargs else "")
        def info(self, event: str, **kwargs: Any) -> None:
            self._base.info("%s %s", event, kwargs if kwargs else "")

        def warning(self, event: str, **kwargs: Any) -> None:
            self._base.warning("%s %s", event, kwargs if kwargs else "")

        def error(self, event: str, **kwargs: Any) -> None:
            self._base.error("%s %s", event, kwargs if kwargs else "")

    logger = _CompatLogger(std_logger)
else:
    logger = structlog.get_logger(__name__)

# Sentinel value for missing arguments
MISSING: object = object()


# ============================================================================
# Base Exceptions and Data Classes
# ============================================================================

class ArgRouterError(RuntimeError):
    """Base exception for routing and validation issues."""


class ArgumentValidationError(ArgRouterError):
    """Raised when the provided payload does not match the method signature."""

    def __init__(
        self,
        class_name: str,
        method_name: str,
        *,
        missing: Iterable[str] | None = None,
        unexpected: Iterable[str] | None = None,
        type_mismatches: Mapping[str, str] | None = None,
    ) -> None:
        self.class_name = class_name
        self.method_name = method_name
        self.missing = set(missing or ())
        self.unexpected = set(unexpected or ())
        self.type_mismatches = dict(type_mismatches or {})
        detail = []
        if self.missing:
            detail.append(f"missing={sorted(self.missing)}")
        if self.unexpected:
            detail.append(f"unexpected={sorted(self.unexpected)}")
        if self.type_mismatches:
            detail.append(f"type_mismatches={self.type_mismatches}")
        message = (
            f"Invalid payload for {class_name}.{method_name}"
            + (f" ({'; '.join(detail)})" if detail else "")
        )
        super().__init__(message)


@dataclass(frozen=True)
class _ParameterSpec:
    name: str
    kind: inspect._ParameterKind
    default: Any
    annotation: Any

    @property
    def required(self) -> bool:
        return self.default is MISSING


@dataclass(frozen=True)
class MethodSpec:
    class_name: str
    method_name: str
    positional: tuple[_ParameterSpec, ...]
    keyword_only: tuple[_ParameterSpec, ...]
    has_var_keyword: bool
    has_var_positional: bool

    @property
    def required_arguments(self) -> tuple[str, ...]:
        required = tuple(
            spec.name
            for spec in (*self.positional, *self.keyword_only)
            if spec.required
        )
        return required

    @property
    def accepted_arguments(self) -> tuple[str, ...]:
        accepted = tuple(spec.name for spec in (*self.positional, *self.keyword_only))
        return accepted


# ============================================================================
# Base ArgRouter (Legacy - use ExtendedArgRouter instead)
# ============================================================================
```

Phase 2 Source Code

```python
class ArgRouter:
    """Resolve method call payloads based on inspected signatures.

    .. note::
        ExtendedArgRouter is the recommended router to use directly.
        This base class is provided for backward compatibility.
    """

    def __init__(self, class_registry: Mapping[str, type]) -> None:
        self._class_registry = dict(class_registry)
        self._spec_cache: dict[tuple[str, str], MethodSpec] = {}
        self._lock = threading.RLock()

    def describe(self, class_name: str, method_name: str) -> MethodSpec:
        """Return the cached method specification, building it if necessary."""
        key = (class_name, method_name)
        with self._lock:
            if key not in self._spec_cache:
                self._spec_cache[key] = self._build_spec(class_name, method_name)
            return self._spec_cache[key]

    def route(
        self,
        class_name: str,
        method_name: str,
        payload: MutableMapping[str, Any],
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """Validate and split a payload into positional and keyword arguments."""
        spec = self.describe(class_name, method_name)
        provided_keys = set(payload.keys())
        required = set(spec.required_arguments)
        accepted = set(spec.accepted_arguments)

        missing = required - provided_keys
        unexpected = provided_keys - accepted
        if unexpected and spec.has_var_keyword:
            unexpected = set()

        if missing or unexpected:
            raise ArgumentValidationError(
                class_name,
                method_name,
                missing=missing,
                unexpected=unexpected,
            )

        args: list[Any] = []
        kwargs: dict[str, Any] = {}
        type_mismatches: dict[str, str] = {}

        remaining = dict(payload)

        for param in spec.positional:
            if param.name not in remaining:
                if param.required:
                    missing = {param.name}
                    raise ArgumentValidationError(
                        class_name,
                        method_name,
                        missing=missing,
                    )
                continue
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            args.append(value)

        for param in spec.keyword_only:
            if param.name not in remaining:
                if param.required:
                    raise ArgumentValidationError(
                        class_name,
                        method_name,
                        missing={param.name},
                    )
                continue
            value = remaining.pop(param.name)
            if not self._matches_annotation(value, param.annotation):
                expected = self._describe_annotation(param.annotation)
                type_mismatches[param.name] = expected
            kwargs[param.name] = value

        if spec.has_var_keyword and remaining:
            kwargs.update(remaining)
            remaining = {}

        if remaining:
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=set(remaining.keys()),
            )

        if type_mismatches:
            raise ArgumentValidationError(
                class_name,
                method_name,
                type_mismatches={
                    name: f"expected {expected}; received {type(payload[name]).__name__}"
                    for name, expected in type_mismatches.items()
                },
            )

        return tuple(args), kwargs

    def expected_arguments(self, class_name: str, method_name: str) -> tuple[str, ...]:
        spec = self.describe(class_name, method_name)
        return spec.accepted_arguments

    def _build_spec(self, class_name: str, method_name: str) -> MethodSpec:
        try:
            cls = self._class_registry[class_name]
        except KeyError as exc:  # pragma: no cover - defensive
            raise ArgRouterError(f"Unknown class '{class_name}'") from exc

        try:
            method = getattr(cls, method_name)
        except AttributeError as exc:
            raise ArgRouterError(f"Class '{class_name}' has no method '{method_name}'") from exc

        signature = inspect.signature(method)
        try:
            type_hints = get_type_hints(method)
        except Exception:
            type_hints = {}
        positional: list[_ParameterSpec] = []
        keyword_only: list[_ParameterSpec] = []
        has_var_keyword = False
        has_var_positional = False

        for parameter in signature.parameters.values():
            if parameter.name == "self":
                continue
            default = (
                parameter.default
                if parameter.default is not inspect._empty
                else MISSING
            )
            annotation = type_hints.get(parameter.name, parameter.annotation)
            param_spec = _ParameterSpec(
                name=parameter.name,
                kind=parameter.kind,
                default=default,
                annotation=annotation,
            )
            if parameter.kind in (
                inspect.Parameter.POSITIONAL_ONLY,
                inspect.Parameter.POSITIONAL_OR_KEYWORD,
            ):
                positional.append(param_spec)
            elif parameter.kind is inspect.Parameter.KEYWORD_ONLY:
                keyword_only.append(param_spec)
            elif parameter.kind is inspect.Parameter.VAR_KEYWORD:
                has_var_keyword = True
            elif parameter.kind is inspect.Parameter.VAR_POSITIONAL:
                has_var_positional = True

        return MethodSpec(
            class_name=class_name,
            method_name=method_name,
            positional=tuple(positional),
            keyword_only=tuple(keyword_only),
            has_var_keyword=has_var_keyword,
            has_var_positional=has_var_positional,
        )

    @staticmethod
    def _matches_annotation(value: Any, annotation: Any) -> bool:
        if annotation in (inspect._empty, Any):
            return True
        origin = get_origin(annotation)
        if origin is None:
            if isinstance(annotation, type):
                return isinstance(value, annotation)
            return True
        args = get_args(annotation)
        if origin is tuple:
            if not isinstance(value, tuple):
                return False
            if not args:
                return True
            if len(args) == 2 and args[1] is Ellipsis:
                return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
            if len(args) != len(value):
                return False
            return all(
                ArgRouter._matches_annotation(item, arg_type)
                for item, arg_type in zip(value, args, strict=False)
            )
        if origin in (list, list):
            if not isinstance(value, list):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (set, set):
            if not isinstance(value, set):
                return False
            if not args:
                return True
            return all(ArgRouter._matches_annotation(item, args[0]) for item in value)
        if origin in (dict, dict):
            if not isinstance(value, dict):
                return False
            if len(args) != 2:
                return True
            key_type, value_type = args
            return all(
                ArgRouter._matches_annotation(k, key_type)
                and ArgRouter._matches_annotation(v, value_type)
                for k, v in value.items()
            )
        if origin is Union:
            return any(ArgRouter._matches_annotation(value, arg) for arg in args)
        return True

    @staticmethod
    def _describe_annotation(annotation: Any) -> str:
        if annotation in (inspect._empty, Any):
            return "Any"
        origin = get_origin(annotation)
        if origin is None:
            if isinstance(annotation, type):
                return annotation.__name__
            return str(annotation)
        args = get_args(annotation)
        if origin is tuple:
            return f"Tuple[{', '.join(ArgRouter._describe_annotation(arg) for arg in args)}]"
        if origin in (list, list):
            return f"List[{ArgRouter._describe_annotation(args[0])}]" if args else "List[Any]"
        if origin in (set, set):
            return f"Set[{ArgRouter._describe_annotation(args[0])}]" if args else "Set[Any]"
        if origin in (dict, dict):
            if len(args) == 2:
                return (
                    f"Dict[{ArgRouter._describe_annotation(args[0])}, "
                    f"{ArgRouter._describe_annotation(args[1])}]"
                )
            return "Dict[Any, Any]"
        if origin is Union:
            return " | ".join(ArgRouter._describe_annotation(arg) for arg in args)
        return str(annotation)


class PayloadDriftMonitor:
    """Sampling validator for ingress/egress payloads."""

    CRITICAL_KEYS = {
        "content": str,
        "pdq_context": (dict, type(None)),
    }

    def __init__(self, *, sample_rate: float, enabled: bool) -> None:
        self.sample_rate = max(0.0, min(sample_rate, 1.0))
        self.enabled = enabled and self.sample_rate > 0.0

    @classmethod
    def from_env(cls) -> PayloadDriftMonitor:
        enabled = os.getenv("ORCHESTRATOR_SAMPLING_VALIDATION", "").lower() in {
            "1",
            "true",
            "yes",
            "on",
        }
        try:
            sample_rate = float(os.getenv("ORCHESTRATOR_SAMPLING_RATE", "0.05"))
        except ValueError:
            sample_rate = 0.05
        return cls(sample_rate=sample_rate, enabled=enabled)

    def maybe_validate(self, payload: Mapping[str, Any], *, producer: str, consumer: str) -> None:
        if not self.enabled:
            return
        if random.random() > self.sample_rate:
            return
        if not isinstance(payload, Mapping):
            return
        keys = set(payload.keys())
        if not keys.intersection(self.CRITICAL_KEYS):
            return

        missing = [key for key in self.CRITICAL_KEYS if key not in payload]
        type_mismatches = {
            key: self._expected_type_name(expected)
            for key, expected in self.CRITICAL_KEYS.items()
            if key in payload and not isinstance(payload[key], expected)
        }
        if missing or type_mismatches:
            std_logger.error(
                "Payload drift detected [%s -> %s]: missing=%s type_mismatches=%s",
                producer,
                consumer,
                missing,
                type_mismatches,
            )
        else:
            std_logger.debug(
                "Payload validation OK [%s -> %s]", producer, consumer
            )

    @staticmethod
    def _expected_type_name(expected: object) -> str:
        if isinstance(expected, tuple):
            return ", ".join(getattr(t, "__name__", str(t)) for t in expected)
        if hasattr(expected, "__name__"):
            return expected.__name__  # type: ignore[arg-type]
        return str(expected)


# ============================================================================
# Extended ArgRouter with Special Routes
# ============================================================================


@dataclass
class RoutingMetrics:
    """Metrics for monitoring routing behavior."""

    total_routes: int = 0
    special_routes_hit: int = 0
    default_routes_hit: int = 0
    validation_errors: int = 0
    silent_drops_prevented: int = 0


class ExtendedArgRouter(ArgRouter):
    """
    Extended argument router with special route handling.

    Extends base ArgRouter with:
    - 25+ special route definitions
    - Strict validation (no silent drops)
    - **kwargs awareness for forward compatibility
    - Comprehensive metrics

    Special Routes (â\211¥25):
    1. _extract_quantitative_claims
    2. _parse_number
    3. _determine_semantic_role
    4. _compile_pattern_registry
    5. _analyze_temporal_coherence
    6. _validate_evidence_chain
    7. _calculate_confidence_score
    8. _extract_indicators
    9. _parse_temporal_reference
    10. _determine_policy_area
    11. _compile_regex_patterns
    12. _analyze_source_reliability
    13. _validate_numerical_consistency
    14. _calculate_bayesian_update
    15. _extract_entities
    16. _parse_citation
    17. _determine_validation_type
    18. _compile_indicator_patterns
    19. _analyze_coherence_score
    20. _validate_threshold_compliance
    21. _calculate_evidence_weight
    22. _extract_temporal_markers
    23. _parse_budget_allocation
    24. _determine_risk_level
    25. _compile_validation_rules
    26. _analyze_stakeholder_impact
    27. _validate_governance_structure
    28. _calculate_alignment_score
    29. _extract_constraint_declarations
    30. _parse_implementation_timeline
    """

    def __init__(self, class_registry: Mapping[str, type]) -> None:
        """
        Initialize extended router.

        Args:
            class_registry: Mapping of class names to class types
        """
        super().__init__(class_registry)
        self._special_routes = self._build_special_routes()
        self._metrics = RoutingMetrics()
        self._metrics_lock = threading.Lock()

        logger.info(
            "extended_arg_router_initialized",
            special_routes=len(self._special_routes),
            classes=len(class_registry),
        )

    def _build_special_routes(self) -> dict[str, dict[str, Any]]:
        """
        Build special route definitions for commonly-called methods.

        Each route specifies:
        - required_args: List of required parameter names
        - optional_args: List of optional parameter names
        - accepts_kwargs: Whether method accepts **kwargs
        - description: Human-readable description

        Returns:
            Dict mapping method names to route specs
        """
        routes = {
            "_extract_quantitative_claims": {
                "required_args": ["content"],
                "optional_args": ["context", "thresholds", "patterns"],
                "accepts_kwargs": True,
                "description": "Extract quantitative claims from content",
            },
            "_parse_number": {
                "required_args": ["text"],
                "optional_args": ["locale", "unit_system"],
                "accepts_kwargs": True,
                "description": "Parse numerical value from text",
            },
            "_determine_semantic_role": {
                "required_args": ["text", "context"],
                "optional_args": ["role_taxonomy", "confidence_threshold"],
                "accepts_kwargs": True,
                "description": "Determine semantic role of text element",
            },
            "_compile_pattern_registry": {
                "required_args": ["patterns"],
                "optional_args": ["category", "flags"],
                "accepts_kwargs": False,
                "description": "Compile patterns into regex registry",
            },
            "_analyze_temporal_coherence": {
                "required_args": ["content"],
                "optional_args": ["temporal_patterns", "baseline_date"],
                "accepts_kwargs": True,
                "description": "Analyze temporal coherence of content",
            },
            "_validate_evidence_chain": {
                "required_args": ["claims", "evidence"],
                "optional_args": ["validation_rules", "min_confidence"],
                "accepts_kwargs": True,
                "description": "Validate evidence chain for claims",
            },
            "_calculate_confidence_score": {
                "required_args": ["evidence"],
                "optional_args": ["prior", "weights"],
                "accepts_kwargs": True,
                "description": "Calculate Bayesian confidence score",
            },
            "_extract_indicators": {
                "required_args": ["content"],
                "optional_args": ["indicator_patterns", "extraction_mode"],
                "accepts_kwargs": True,
                "description": "Extract KPI indicators from content",
            },
            "_parse_temporal_reference": {
                "required_args": ["text"],
                "optional_args": ["reference_date", "format_hints"],
                "accepts_kwargs": True,
                "description": "Parse temporal reference from text",
            },
            "_determine_policy_area": {
                "required_args": ["content"],
                "optional_args": ["taxonomy", "multi_label"],
                "accepts_kwargs": True,
                "description": "Classify content into policy area",
            },
            "_compile_regex_patterns": {
                "required_args": ["pattern_list"],
                "optional_args": ["flags", "validate"],
                "accepts_kwargs": False,
                "description": "Compile list of regex patterns",
            },
            "_analyze_source_reliability": {
                "required_args": ["source"],
                "optional_args": ["source_patterns", "reliability_threshold"],
                "accepts_kwargs": True,
                "description": "Analyze reliability of information source",
            },
            "_validate_numerical_consistency": {
                "required_args": ["numbers"],
                "optional_args": ["tolerance", "consistency_rules"],
                "accepts_kwargs": True,
                "description": "Validate numerical consistency across values",
            },
            "_calculate_bayesian_update": {
                "required_args": ["prior", "likelihood", "evidence"],
                "optional_args": ["normalization"],
                "accepts_kwargs": True,
                "description": "Calculate Bayesian posterior update",
            },
            "_extract_entities": {
                "required_args": ["content"],
                "optional_args": ["entity_types", "confidence_threshold"],
                "accepts_kwargs": True,
                "description": "Extract named entities from content",
            },
            "_parse_citation": {
                "required_args": ["text"],
                "optional_args": ["citation_style", "strict_mode"],
                "accepts_kwargs": True,
                "description": "Parse citation from text",
            },
            "_determine_validation_type": {
                "required_args": ["validation_spec"],
                "optional_args": ["context"],
                "accepts_kwargs": True,
                "description": "Determine type of validation to apply",
            },
            "_compile_indicator_patterns": {
                "required_args": ["indicators"],
                "optional_args": ["category", "weights"],
                "accepts_kwargs": False,
                "description": "Compile indicator patterns for matching",
            },
            "_analyze_coherence_score": {
                "required_args": ["content"],
                "optional_args": ["coherence_patterns", "scoring_mode"],
                "accepts_kwargs": True,
                "description": "Analyze narrative coherence score",
            },
            "_validate_threshold_compliance": {
                "required_args": ["value", "thresholds"],
                "optional_args": ["strict_mode"],
                "accepts_kwargs": True,
                "description": "Validate value against thresholds",
            },
            "_calculate_evidence_weight": {
                "required_args": ["evidence"],
                "optional_args": ["weighting_scheme", "normalization"],
                "accepts_kwargs": True,
                "description": "Calculate evidence weight for scoring",
            },
            "_extract_temporal_markers": {
                "required_args": ["content"],
                "optional_args": ["temporal_patterns", "extraction_depth"],
                "accepts_kwargs": True,
                "description": "Extract temporal markers from content",
            },
            "_parse_budget_allocation": {
                "required_args": ["text"],
                "optional_args": ["currency", "fiscal_year"],
                "accepts_kwargs": True,
                "description": "Parse budget allocation from text",
            },
            "_determine_risk_level": {
                "required_args": ["indicators"],
                "optional_args": ["risk_thresholds", "aggregation_method"],
                "accepts_kwargs": True,
                "description": "Determine risk level from indicators",
            },
            "_compile_validation_rules": {
                "required_args": ["rules"],
                "optional_args": ["rule_format"],
                "accepts_kwargs": False,
                "description": "Compile validation rules for execution",
            },
            "_analyze_stakeholder_impact": {
                "required_args": ["stakeholders", "policy"],
                "optional_args": ["impact_dimensions", "time_horizon"],
                "accepts_kwargs": True,
                "description": "Analyze stakeholder impact of policy",
            },
            "_validate_governance_structure": {
                "required_args": ["structure"],
                "optional_args": ["governance_standards", "strict_mode"],
                "accepts_kwargs": True,
                "description": "Validate governance structure compliance",
            },
            "_calculate_alignment_score": {
                "required_args": ["policy_content", "reference_framework"],
                "optional_args": ["alignment_weights", "scoring_method"],
                "accepts_kwargs": True,
                "description": "Calculate alignment score with framework",
            },
            "_extract_constraint_declarations": {
                "required_args": ["content"],
                "optional_args": ["constraint_types", "extraction_mode"],
                "accepts_kwargs": True,
                "description": "Extract constraint declarations from content",
            },
            "_parse_implementation_timeline": {
                "required_args": ["text"],
                "optional_args": ["reference_date", "granularity"],
                "accepts_kwargs": True,
                "description": "Parse implementation timeline from text",
            },
        }

        return routes

    def route(
        self,
        class_name: str,
        method_name: str,
        payload: MutableMapping[str, Any],
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """
        Route method call with special handling and strict validation.

        This override:
        1. Checks for special route definitions
        2. Applies strict validation
        3. Prevents silent parameter drops
        4. Tracks metrics

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Method parameters

        Returns:
            Tuple of (args, kwargs) for method invocation

        Raises:
            ArgumentValidationError: On validation failure
        """
        with self._metrics_lock:
            self._metrics.total_routes += 1

        # Check for special route
        if method_name in self._special_routes:
            return self._route_special(class_name, method_name, payload)

        # Use default routing with enhanced validation
        return self._route_default_strict(class_name, method_name, payload)

    def _route_special(
        self,
        class_name: str,
        method_name: str,
        payload: MutableMapping[str, Any],
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """
        Route using special route definition.

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Method parameters

        Returns:
            Tuple of (args, kwargs)
        """
        with self._metrics_lock:
            self._metrics.special_routes_hit += 1

        route_spec = self._special_routes[method_name]
        required_args = set(route_spec["required_args"])
        optional_args = set(route_spec["optional_args"])
        accepts_kwargs = route_spec["accepts_kwargs"]

        provided_keys = set(payload.keys())

        # Check required arguments
        missing = required_args - provided_keys
        if missing:
            with self._metrics_lock:
                self._metrics.validation_errors += 1
            logger.error(
                "special_route_missing_args",
                class_name=class_name,
                method=method_name,
                missing=sorted(missing),
            )
            raise ArgumentValidationError(
                class_name,
                method_name,
                missing=missing,
            )

        # Check unexpected arguments
        expected = required_args | optional_args
        unexpected = provided_keys - expected

        if unexpected and not accepts_kwargs:
            # Method doesn’t accept **kwargs, so unexpected args are an error
            with self._metrics_lock:
                self._metrics.validation_errors += 1
                self._metrics.silent_drops_prevented += 1

            logger.error(
                "special_route_unexpected_args",
                class_name=class_name,
                method=method_name,
                unexpected=sorted(unexpected),
                accepts_kwargs=accepts_kwargs,
            )
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=unexpected,
            )

        # Build kwargs (all parameters go to kwargs for special routes)
        kwargs = dict(payload)

        logger.debug(
            "special_route_applied",
            class_name=class_name,
            method=method_name,
            params_count=len(kwargs),
        )

        return (), kwargs

    def _route_default_strict(
        self,
        class_name: str,
        method_name: str,
        payload: MutableMapping[str, Any],
    ) -> tuple[tuple[Any, ...], dict[str, Any]]:
        """
        Route using default strategy with strict validation.

        This prevents silent parameter drops by failing when:
        - Required arguments are missing
        - Unexpected arguments are provided AND method lacks **kwargs

        Args:
            class_name: Target class name
            method_name: Target method name
            payload: Method parameters

        Returns:
            Tuple of (args, kwargs)
        """
        with self._metrics_lock:
            self._metrics.default_routes_hit += 1

        # Use base implementation for inspection
        spec = self.describe(class_name, method_name)

        # Strict validation: if unexpected args and no **kwargs, fail
        provided_keys = set(payload.keys())
        accepted = set(spec.accepted_arguments)
        unexpected = provided_keys - accepted

        if unexpected and not spec.has_var_keyword:
            # Method doesn’t accept **kwargs - unexpected args are errors
            with self._metrics_lock:
                self._metrics.validation_errors += 1
                self._metrics.silent_drops_prevented += 1

            logger.error(
                "default_route_unexpected_args_strict",
                class_name=class_name,
                method=method_name,
                unexpected=sorted(unexpected),
                has_var_keyword=spec.has_var_keyword,
            )
            raise ArgumentValidationError(
                class_name,
                method_name,
                unexpected=unexpected,
            )

        # Delegate to base implementation
        try:
            result = super().route(class_name, method_name, payload)
            logger.debug(
                "default_route_applied",
                class_name=class_name,
                method=method_name,
            )
            return result
        except ArgumentValidationError:
            with self._metrics_lock:
                self._metrics.validation_errors += 1
            raise

    def get_special_route_coverage(self) -> int:
        """
        Get count of special routes defined.
        Returns:
            Number of special routes (target: â\211¥25)
        """
        return len(self._special_routes)

    def get_metrics(self) -> dict[str, Any]:
        """
        Get routing metrics.

        Returns:
            Dict with routing statistics
        """
        total = self._metrics.total_routes or 1  # Avoid division by zero

        return {
            "total_routes": self._metrics.total_routes,
            "special_routes_hit": self._metrics.special_routes_hit,
            "special_routes_coverage": len(self._special_routes),
            "default_routes_hit": self._metrics.default_routes_hit,
            "validation_errors": self._metrics.validation_errors,
            "silent_drops_prevented": self._metrics.silent_drops_prevented,
            "special_route_hit_rate": self._metrics.special_routes_hit / total,
            "error_rate": self._metrics.validation_errors / total,
        }

    def list_special_routes(self) -> list[dict[str, Any]]:
        """
        List all special routes with their specifications.

        Returns:
            List of route specifications
        """
        routes = []
        for method_name, spec in sorted(self._special_routes.items()):
            routes.append({
                "method_name": method_name,
                "required_args": spec["required_args"],
                "optional_args": spec["optional_args"],
                "accepts_kwargs": spec["accepts_kwargs"],
                "description": spec["description"],
            })
        return routes



################################################################################
# FILE: phase2_aa_method_source_validator.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_aa_method_source_validator.py
################################################################################

"""
Module: phase2_aa_method_source_validator
PHASE_LABEL: Phase 2
Sequence: AA

"""

import ast
import os
import json
from typing import Dict, List, Any

class MethodSourceValidator:
    def __init__(self, base_path: str = "src/farfan_pipeline"):
        self.base_path = base_path
        self.source_map = self._build_source_map()

    def _build_source_map(self) -> Dict[str, Dict[str, Any]]:
        class_map = {}
        for root, _, files in os.walk(self.base_path):
            for file in files:
                if file.endswith(".py"):
                    file_path = os.path.join(root, file)
                    with open(file_path, "r", encoding="utf-8") as f:
                        try:
                            tree = ast.parse(f.read(), filename=file_path)
                            for node in ast.walk(tree):
                                if isinstance(node, ast.ClassDef):
                                    class_name = node.name
                                    methods = []
                                    for item in node.body:
                                        if isinstance(item, ast.FunctionDef):
                                            methods.append(item.name)

                                    if class_name in class_map:
                                        # In case of duplicate class names, we might need a more robust way
                                        # to handle this, but for now we’ll just overwrite.
                                        # A better approach could be to store a list of locations.
                                        pass

                                    class_map[class_name] = {
                                        "file_path": file_path,
                                        "methods": methods,
                                    }
                        except Exception as e:
                            print(f"Error parsing {file_path}: {e}")
        return class_map

    def validate_executor_methods(self, executor_methods_path: str = "src/farfan_pipeline/core/orchestrator/executors_methods.j
son") -> Dict[str, List[str]]:
        with open(executor_methods_path, "r") as f:
            executor_data = json.load(f)

        declared_methods = set()
        for executor_info in executor_data:
            for method_info in executor_info.get("methods", []):
                class_name = method_info.get("class")
                method_name = method_info.get("method")
                if class_name and method_name:
                    declared_methods.add(f"{class_name}.{method_name}")

        valid = []
        missing = []

        for method_fqn in declared_methods:
            if "." not in method_fqn:
                # Assuming methods are always Class.method
                continue

            class_name, method_name = method_fqn.split(".", 1)

            if class_name not in self.source_map:
                missing.append(method_fqn)
                continue

            class_info = self.source_map[class_name]
            if method_name not in class_info["methods"]:
                missing.append(method_fqn)
            else:
                valid.append(method_fqn)

        # Phantom methods would be those in source but not declared.
        # The user’s request seems to focus on missing/valid from declaration.
        # "phantom" is defined by user as "executors call fantasy methods"
        # which is covered by "missing"
        return {"valid": valid, "missing": missing, "phantom": []}


    def generate_source_truth_map(self) -> Dict[str, Dict[str, Any]]:
        source_truth = {}
        for class_name, info in self.source_map.items():
            file_path = info["file_path"]
            with open(file_path, "r", encoding="utf-8") as f:
                tree = ast.parse(f.read(), filename=file_path)
                for node in ast.walk(tree):
                    if isinstance(node, ast.ClassDef) and node.name == class_name:
                        for item in node.body:
                            if isinstance(item, ast.FunctionDef):
                                method_name = item.name
                                fqn = f"{class_name}.{method_name}"

                                # Basic signature extraction
                                args = [arg.arg for arg in item.args.args]
                                signature = f"({’, ’.join(args)})"
                                # A more advanced version would parse type hints if they exist

                                source_truth[fqn] = {
                                    "exists": True,
                                    "file": file_path,
                                    "line": item.lineno,
                                    "signature": signature,
                                }
        return source_truth

if __name__ == "__main__":
    validator = MethodSourceValidator()

    # 1. Generate the ground-truth map
    source_truth_map = validator.generate_source_truth_map()
    output_path = "method_source_truth.json"
    with open(output_path, "w") as f:
        json.dump(source_truth_map, f, indent=4)
    print(f"Generated source truth map at {output_path}")

    # 2. Validate executor methods
    validation_report = validator.validate_executor_methods()
    report_path = "executor_validation_report.json"
    with open(report_path, "w") as f:
        json.dump(validation_report, f, indent=4)
    print(f"Validation report generated at {report_path}")

    print("\nValidation Summary:")
    print(f"  - Valid methods: {len(validation_report[’valid’])}")
    print(f"  - Missing methods: {len(validation_report[’missing’])}")
    if validation_report[’missing’]:
        print("\nMissing methods:")
        for method in validation_report[’missing’]:
            print(f"  - {method}")
```

Phase 2 Source Code

```
################################################################################
# FILE: phase2_ab_resource_alerts.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_ab_resource_alerts.py
################################################################################

"""
Module: phase2_ab_resource_alerts
PHASE_LABEL: Phase 2
Sequence: AB

"""
"""Resource Pressure Alerting and Observability.

Provides comprehensive alerting and monitoring for resource management:
- Structured logging for resource events
- Alert thresholds and notifications
- Integration with external monitoring systems
- Historical trend analysis
"""

from __future__ import annotations

import json
import logging
from collections import defaultdict
from datetime import datetime, timedelta
from enum import Enum
from typing import Any, Callable

from orchestration.resource_manager import (
    ResourcePressureEvent,
    ResourcePressureLevel,
)

logger = logging.getLogger(__name__)


class AlertSeverity(Enum):
    """Alert severity levels."""

    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
    CRITICAL = "critical"


class AlertChannel(Enum):
    """Alert delivery channels."""

    LOG = "log"
    WEBHOOK = "webhook"
    SIGNAL = "signal"
    STDOUT = "stdout"


class ResourceAlert:
    """Individual resource alert."""

    def __init__(
        self,
        severity: AlertSeverity,
        title: str,
        message: str,
        event: ResourcePressureEvent,
        metadata: dict[str, Any] | None = None,
    ) -> None:
        self.severity = severity
        self.title = title
        self.message = message
        self.event = event
        self.metadata = metadata or {}
        self.timestamp = datetime.utcnow()
        self.alert_id = f"alert_{self.timestamp.isoformat()}_{id(self)}"

    def to_dict(self) -> dict[str, Any]:
        """Convert alert to dictionary."""
        return {
            "alert_id": self.alert_id,
            "timestamp": self.timestamp.isoformat(),
            "severity": self.severity.value,
            "title": self.title,
            "message": self.message,
            "event": {
                "timestamp": self.event.timestamp.isoformat(),
                "pressure_level": self.event.pressure_level.value,
                "cpu_percent": self.event.cpu_percent,
                "memory_mb": self.event.memory_mb,
                "memory_percent": self.event.memory_percent,
                "worker_count": self.event.worker_count,
                "active_executors": self.event.active_executors,
                "degradation_applied": self.event.degradation_applied,
                "circuit_breakers_open": self.event.circuit_breakers_open,
            },
            "metadata": self.metadata,
        }
```

Phase 2 Source Code

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


class AlertThresholds:
    """Configurable alert thresholds."""

    def __init__(
        self,
        memory_warning_percent: float = 75.0,
        memory_critical_percent: float = 85.0,
        cpu_warning_percent: float = 75.0,
        cpu_critical_percent: float = 85.0,
        circuit_breaker_warning_count: int = 3,
        degradation_critical_count: int = 3,
    ) -> None:
        self.memory_warning_percent = memory_warning_percent
        self.memory_critical_percent = memory_critical_percent
        self.cpu_warning_percent = cpu_warning_percent
        self.cpu_critical_percent = cpu_critical_percent
        self.circuit_breaker_warning_count = circuit_breaker_warning_count
        self.degradation_critical_count = degradation_critical_count


class ResourceAlertManager:
    """Manages resource pressure alerts and notifications."""

    def __init__(
        self,
        thresholds: AlertThresholds | None = None,
        channels: list[AlertChannel] | None = None,
        webhook_url: str | None = None,
        signal_callback: Callable[[ResourceAlert], None] | None = None,
    ) -> None:
        self.thresholds = thresholds or AlertThresholds()
        self.channels = channels or [AlertChannel.LOG]
        self.webhook_url = webhook_url
        self.signal_callback = signal_callback

        self.alert_history: list[ResourceAlert] = []
        self.alert_counts: dict[str, int] = defaultdict(int)
        self.suppressed_alerts: set[str] = set()
        self.last_alert_times: dict[str, datetime] = {}

    def process_event(self, event: ResourcePressureEvent) -> list[ResourceAlert]:
        """Process resource pressure event and generate alerts."""
        alerts: list[ResourceAlert] = []

        memory_alert = self._check_memory_threshold(event)
        if memory_alert:
            alerts.append(memory_alert)

        cpu_alert = self._check_cpu_threshold(event)
        if cpu_alert:
            alerts.append(cpu_alert)

        pressure_alert = self._check_pressure_level(event)
        if pressure_alert:
            alerts.append(pressure_alert)

        circuit_breaker_alert = self._check_circuit_breakers(event)
        if circuit_breaker_alert:
            alerts.append(circuit_breaker_alert)

        degradation_alert = self._check_degradation(event)
        if degradation_alert:
            alerts.append(degradation_alert)

        for alert in alerts:
            self._dispatch_alert(alert)
            self.alert_history.append(alert)
            self.alert_counts[alert.severity.value] += 1

        return alerts

    def _check_memory_threshold(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if memory usage exceeds thresholds."""
        if event.memory_percent >= self.thresholds.memory_critical_percent:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Critical Memory Usage",
                message=f"Memory usage at {event.memory_percent:.1f}% "
                f"({event.memory_mb:.1f} MB)",
                event=event,
                metadata={"threshold": self.thresholds.memory_critical_percent},
            )

        if event.memory_percent >= self.thresholds.memory_warning_percent:
            if self._should_alert("memory_warning", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="High Memory Usage",
                    message=f"Memory usage at {event.memory_percent:.1f}% "
                f"({event.memory_mb:.1f} MB)",
                event=event,
                metadata={"threshold": self.thresholds.memory_warning_percent},
            )

        return None

    def _check_cpu_threshold(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if CPU usage exceeds thresholds."""
        if event.cpu_percent >= self.thresholds.cpu_critical_percent:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Critical CPU Usage",
                message=f"CPU usage at {event.cpu_percent:.1f}%",
                event=event,
                metadata={"threshold": self.thresholds.cpu_critical_percent},
            )

        if event.cpu_percent >= self.thresholds.cpu_warning_percent:
            if self._should_alert("cpu_warning", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="High CPU Usage",
                    message=f"CPU usage at {event.cpu_percent:.1f}%",
                    event=event,
                    metadata={"threshold": self.thresholds.cpu_warning_percent},
                )

        return None

    def _check_pressure_level(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if pressure level warrants alert."""
        if event.pressure_level == ResourcePressureLevel.EMERGENCY:
            return ResourceAlert(
                severity=AlertSeverity.CRITICAL,
                title="Emergency Resource Pressure",
                message="System under emergency resource pressure",
                event=event,
            )

        if event.pressure_level == ResourcePressureLevel.CRITICAL:
            if self._should_alert("pressure_critical", minutes=2):
                return ResourceAlert(
                    severity=AlertSeverity.ERROR,
                    title="Critical Resource Pressure",
                    message="System under critical resource pressure",
                    event=event,
                )

        if event.pressure_level == ResourcePressureLevel.HIGH:
            if self._should_alert("pressure_high", minutes=10):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="High Resource Pressure",
                    message="System experiencing high resource pressure",
                    event=event,
                )

        return None

    def _check_circuit_breakers(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if circuit breakers warrant alert."""
        open_count = len(event.circuit_breakers_open)

        if open_count >= self.thresholds.circuit_breaker_warning_count:
            return ResourceAlert(
                severity=AlertSeverity.ERROR,
                title="Multiple Circuit Breakers Open",
                message=f"{open_count} circuit breakers are open: "
                f"{', '.join(event.circuit_breakers_open)}",
                event=event,
                metadata={
                    "open_count": open_count,
                    "executors": event.circuit_breakers_open,
                },
            )

        if open_count > 0:
            if self._should_alert("circuit_breaker", minutes=5):
                return ResourceAlert(
                    severity=AlertSeverity.WARNING,
                    title="Circuit Breaker Opened",
                    message=f"Circuit breakers open for: "
                    f"{', '.join(event.circuit_breakers_open)}",
                    event=event,
                    metadata={"executors": event.circuit_breakers_open},
                )

        return None
```

Phase 2 Source Code

```
    def _check_degradation(
        self, event: ResourcePressureEvent
    ) -> ResourceAlert | None:
        """Check if degradation strategies warrant alert."""
        degradation_count = len(event.degradation_applied)

        if degradation_count >= self.thresholds.degradation_critical_count:
            return ResourceAlert(
                severity=AlertSeverity.ERROR,
                title="Multiple Degradation Strategies Active",
                message=f"{degradation_count} degradation strategies applied: "
                f"{', '.join(event.degradation_applied)}",
                event=event,
                metadata={
                    "count": degradation_count,
                    "strategies": event.degradation_applied,
                },
            )

        if degradation_count > 0:
            if self._should_alert("degradation", minutes=10):
                return ResourceAlert(
                    severity=AlertSeverity.INFO,
                    title="Degradation Strategies Active",
                    message=f"Active degradation: "
                    f"{', '.join(event.degradation_applied)}",
                    event=event,
                    metadata={"strategies": event.degradation_applied},
                )

        return None

    def _should_alert(self, alert_type: str, minutes: int = 5) -> bool:
        """Check if alert should be sent (with rate limiting)."""
        now = datetime.utcnow()
        last_time = self.last_alert_times.get(alert_type)

        if not last_time:
            self.last_alert_times[alert_type] = now
            return True

        elapsed = (now - last_time).total_seconds() / 60
        if elapsed >= minutes:
            self.last_alert_times[alert_type] = now
            return True

        return False

    def _dispatch_alert(self, alert: ResourceAlert) -> None:
        """Dispatch alert to configured channels."""
        for channel in self.channels:
            try:
                if channel == AlertChannel.LOG:
                    self._log_alert(alert)
                elif channel == AlertChannel.WEBHOOK:
                    self._send_webhook(alert)
                elif channel == AlertChannel.SIGNAL:
                    self._send_signal(alert)
                elif channel == AlertChannel.STDOUT:
                    self._print_alert(alert)
            except Exception as exc:
                logger.error(
                    f"Failed to dispatch alert to {channel.value}: {exc}"
                )

    def _log_alert(self, alert: ResourceAlert) -> None:
        """Log alert with appropriate severity."""
        extra = {
            "alert_id": alert.alert_id,
            "alert_severity": alert.severity.value,
            "pressure_level": alert.event.pressure_level.value,
            "cpu_percent": alert.event.cpu_percent,
            "memory_mb": alert.event.memory_mb,
        }

        if alert.severity == AlertSeverity.CRITICAL:
            logger.critical(f"{alert.title}: {alert.message}", extra=extra)
        elif alert.severity == AlertSeverity.ERROR:
            logger.error(f"{alert.title}: {alert.message}", extra=extra)
        elif alert.severity == AlertSeverity.WARNING:
            logger.warning(f"{alert.title}: {alert.message}", extra=extra)
        else:
            logger.info(f"{alert.title}: {alert.message}", extra=extra)

    def _send_webhook(self, alert: ResourceAlert) -> None:
        """Send alert via webhook."""
        if not self.webhook_url:
            return

        try:
            import requests

            requests.post(
                self.webhook_url,
                json=alert.to_dict(),
                timeout=5,
            )
        except Exception as exc:
            logger.error(f"Webhook alert failed: {exc}")

    def _send_signal(self, alert: ResourceAlert) -> None:
        """Send alert via signal callback."""
        if not self.signal_callback:
            return

        try:
            self.signal_callback(alert)
        except Exception as exc:
            logger.error(f"Signal callback failed: {exc}")

    def _print_alert(self, alert: ResourceAlert) -> None:
        """Print alert to stdout."""
        severity_colors = {
            AlertSeverity.INFO: "\033[94m",
            AlertSeverity.WARNING: "\033[93m",
            AlertSeverity.ERROR: "\033[91m",
            AlertSeverity.CRITICAL: "\033[95m",
        }
        reset = "\033[0m"

        color = severity_colors.get(alert.severity, reset)
        print(
            f"{color}[{alert.severity.value.upper()}] {alert.title}: "
            f"{alert.message}{reset}"
        )

    def get_alert_summary(self) -> dict[str, Any]:
        """Get summary of alert history."""
        now = datetime.utcnow()
        hour_ago = now - timedelta(hours=1)
        day_ago = now - timedelta(days=1)

        recent_alerts = [
            alert for alert in self.alert_history if alert.timestamp >= hour_ago
        ]

        daily_alerts = [
            alert for alert in self.alert_history if alert.timestamp >= day_ago
        ]

        return {
            "total_alerts": len(self.alert_history),
            "last_hour": len(recent_alerts),
            "last_24_hours": len(daily_alerts),
            "by_severity": dict(self.alert_counts),
            "recent_alerts": [alert.to_dict() for alert in recent_alerts[-10:]],
        }

    def clear_history(self) -> None:
        """Clear alert history."""
        self.alert_history.clear()
        self.alert_counts.clear()
        self.last_alert_times.clear()



################################################################################
# FILE: phase2_b_base_executor_with_contract.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_b_base_executor_with_contract.py
################################################################################

"""Base Executor with Contract-driven execution.

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Base Executor
PHASE_ROLE: Abstract base class for contract-driven executors with method routing

from __future__ import annotations

import json
from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, Any

try:
    from jsonschema import Draft7Validator  # type: ignore
except Exception:  # pragma: no cover
    Draft7Validator = Any  # type: ignore[misc,assignment]

from canonic_phases.Phase_zero.paths import PROJECT_ROOT
# NEW: Replace legacy evidence modules with EvidenceNexus and Carver
from canonic_phases.Phase_two.evidence_nexus import EvidenceNexus, process_evidence
from canonic_phases.Phase_two.carver import DoctoralCarverSynthesizer
from canonic_phases.Phase_two.calibration_policy import CalibrationPolicy, create_default_policy

if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor
    from farfan_pipeline.core.types import PreprocessedDocument
else:  # pragma: no cover - runtime avoids import to break cycles
    MethodExecutor = Any
    PreprocessedDocument = Any
```

Phase 2 Source Code

```
class BaseExecutorWithContract(ABC):
    """Contract-driven executor that routes all calls through MethodExecutor.

    Supports both v2 and v3 contract formats:
    - v2: Legacy format with method_inputs, assembly_rules, validation_rules at top level
    - v3: New format with identity, executor_binding, method_binding, question_context,
          evidence_assembly, output_contract, validation_rules, etc.

    Contract version is auto-detected based on file name (.v3.json vs .json) and structure.
    """

    _contract_cache: dict[str, dict[str, Any]] = {}
    _schema_validators: dict[str, Draft7Validator] = {}
    _factory_contracts_verified: bool = False
    _factory_verification_errors: list[str] = []

    def __init__(
        self,
        method_executor: MethodExecutor,
        signal_registry: Any,
        config: Any,
        questionnaire_provider: Any,
        calibration_orchestrator: Any | None = None,
        enriched_packs: dict[str, Any] | None = None,
        validation_orchestrator: Any | None = None,
        calibration_policy: CalibrationPolicy | None = None,
    ) -> None:
        self.method_executor = method_executor
        self.signal_registry = signal_registry
        self.config = config
        self.questionnaire_provider = questionnaire_provider
        self.calibration_orchestrator = calibration_orchestrator
        # JOBFRONT 3: Support for enriched signal packs (intelligence layer)
        self.enriched_packs = enriched_packs or {}
        self._use_enriched_signals = len(self.enriched_packs) > 0
        # VALIDATION ORCHESTRATOR: Comprehensive validation tracking
        self.validation_orchestrator = validation_orchestrator
        self._use_validation_orchestrator = validation_orchestrator is not None
        # CALIBRATION POLICY: Method selection and weighting based on calibration
        self.calibration_policy = calibration_policy or create_default_policy(strict_mode=False)

    @classmethod
    @abstractmethod
    def get_base_slot(cls) -> str:
        raise NotImplementedError

    @classmethod
    def verify_all_base_contracts(
        cls, class_registry: dict[str, type[object]] | None = None
    ) -> dict[str, Any]:
        """Verify all 30 base executor contracts at factory initialization time.

        This method loads and validates all contracts for D1-Q1 through D6-Q5, checking:
        - Contract files exist and are valid JSON
        - Required fields are present (method_inputs/method_binding, assembly_rules,
          validation_rules, expected_elements)
        - JSON schema compliance (v2 or v3)
        - All referenced method classes exist in the class registry

        Args:
            class_registry: Optional class registry to verify method class existence.
                           If None, will attempt to import and build one.

        Returns:
            dict with keys:
                - passed: bool indicating if all contracts are valid
                - total_contracts: int count of contracts checked
                - errors: list of error messages for failed contracts
                - warnings: list of warning messages
                - verified_contracts: list of base_slot identifiers that passed

        Raises:
            RuntimeError: If verification fails with strict=True
        """
        if cls._factory_contracts_verified:
            return {
                "passed": len(cls._factory_verification_errors) == 0,
                "total_contracts": 30,
                "errors": cls._factory_verification_errors,
                "warnings": [],
                "verified_contracts": list(cls._contract_cache.keys()),
            }

        base_slots = [
            f"D{d}-Q{q}" for d in range(1, 7) for q in range(1, 6)
        ]

        if class_registry is None:
            try:
                from orchestration.class_registry import (
                    build_class_registry,
                )
                class_registry = build_class_registry()
            except Exception as exc:
                cls._factory_verification_errors.append(
                    f"Failed to build class registry for verification: {exc}"
            )

        errors: list[str] = []
        warnings: list[str] = []
        verified_contracts: list[str] = []

        for base_slot in base_slots:
            try:
                result = cls._verify_single_contract(base_slot, class_registry)
                if result["passed"]:
                    verified_contracts.append(base_slot)
                else:
                    errors.extend(
                        f"[{base_slot}] {err}" for err in result["errors"]
                    )
                warnings.extend(
                    f"[{base_slot}] {warn}" for warn in result.get("warnings", [])
                )
            except Exception as exc:
                errors.append(f"[{base_slot}] Unexpected error during verification: {exc}")

        cls._factory_contracts_verified = True
        cls._factory_verification_errors = errors

        return {
            "passed": len(errors) == 0,
            "total_contracts": len(base_slots),
            "errors": errors,
            "warnings": warnings,
            "verified_contracts": verified_contracts,
        }

    @classmethod
    def _verify_single_contract(
        cls, base_slot: str, class_registry: dict[str, type[object]] | None = None
    ) -> dict[str, Any]:
        """Verify a single contract for completeness and validity.

        Args:
            base_slot: Base slot identifier (e.g., "D1-Q1")
            class_registry: Optional class registry for method class verification

        Returns:
            dict with keys:
                - passed: bool
                - errors: list of error messages
                - warnings: list of warning messages
                - contract_version: detected version (v2/v3)
                - contract_path: path to contract file
        """
        errors: list[str] = []
        warnings: list[str] = []

        dimension = int(base_slot[1])
        question = int(base_slot[4])
        q_number = (dimension - 1) * 5 + question
        q_id = f"Q{q_number:03d}"

        contracts_dir = PROJECT_ROOT / "src" / "farfan_pipeline" / "phases" / "Phase_two" / "json_files_phase_two" / "executor_
contracts"

        v3_path = contracts_dir / f"{base_slot}.v3.json"
        v2_path = contracts_dir / f"{base_slot}.json"
        v3_specialized_path = contracts_dir / "specialized" / f"{q_id}.v3.json"
        v2_specialized_path = contracts_dir / "specialized" / f"{q_id}.json"

        contract_path = None
        if v3_path.exists():
            contract_path = v3_path
            expected_version = "v3"
        elif v2_path.exists():
            contract_path = v2_path
            expected_version = "v2"
        elif v3_specialized_path.exists():
            contract_path = v3_specialized_path
            expected_version = "v3"
        elif v2_specialized_path.exists():
            contract_path = v2_specialized_path
            expected_version = "v2"
        else:
            errors.append(
                f"Contract file not found. Tried: {v3_path}, {v2_path}, {v3_specialized_path}, {v2_specialized_path}"
            )
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": None,
                "contract_path": None,
            }

        try:
            contract = json.loads(contract_path.read_text(encoding="utf-8"))
        except json.JSONDecodeError as exc:
            errors.append(f"Invalid JSON in contract file: {exc}")
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": expected_version,
                "contract_path": str(contract_path),
            }
        except Exception as exc:
            errors.append(f"Failed to read contract file: {exc}")
            return {
                "passed": False,
                "errors": errors,
                "warnings": warnings,
                "contract_version": expected_version,
                "contract_path": str(contract_path),
            }

        detected_version = cls._detect_contract_version(contract)
        if detected_version != expected_version:
            warnings.append(
                f"Contract structure is {detected_version} but file naming suggests {expected_version}"
            )

        try:
            validator = cls._get_schema_validator(detected_version)
            schema_errors = sorted(validator.iter_errors(contract), key=lambda e: e.path)
            if schema_errors:
                errors.extend(
                    f"Schema validation error: {err.message} at {'.'.join(str(p) for p in err.path)}"
                    for err in schema_errors[:10]
                )
        except FileNotFoundError as exc:
            warnings.append(f"Schema file not found: {exc}. Skipping schema validation.")
        except Exception as exc:
            warnings.append(f"Schema validation error: {exc}")

        if detected_version == "v3":
            v3_errors = cls._verify_v3_contract_fields(contract, base_slot, class_registry)
            errors.extend(v3_errors)
        else:
            v2_errors = cls._verify_v2_contract_fields(contract, base_slot, class_registry)
            errors.extend(v2_errors)

        return {
            "passed": len(errors) == 0,
            "errors": errors,
            "warnings": warnings,
            "contract_version": detected_version,
            "contract_path": str(contract_path),
        }

    @classmethod
    def _verify_v2_contract_fields(
        cls,
        contract: dict[str, Any],
        base_slot: str,
        class_registry: dict[str, type[object]] | None = None,
    ) -> list[str]:
        """Verify required fields for v2 contract format.

        Args:
            contract: Parsed contract dict
            base_slot: Base slot identifier
            class_registry: Optional class registry for method verification

        Returns:
            List of error messages (empty if all checks pass)
        """
        errors: list[str] = []

        if "method_inputs" not in contract:
            errors.append("Missing required field: method_inputs")
        elif not isinstance(contract["method_inputs"], list):
            errors.append("method_inputs must be a list")
        else:
            method_inputs = contract["method_inputs"]
            if not method_inputs:
                errors.append("method_inputs is empty")
            else:
                for idx, method_spec in enumerate(method_inputs):
                    if not isinstance(method_spec, dict):
                        errors.append(f"method_inputs[{idx}] is not a dict")
                        continue
                    if "class" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'class' field")
                    if "method" not in method_spec:
                        errors.append(f"method_inputs[{idx}] missing 'method' field")

                    if class_registry is not None and "class" in method_spec:
                        class_name = method_spec["class"]
                        if class_name not in class_registry:
                            errors.append(
                                f"method_inputs[{idx}]: class '{class_name}' not found in class registry"
                            )

        if "assembly_rules" not in contract:
            errors.append("Missing required field: assembly_rules")
        elif not isinstance(contract["assembly_rules"], list):
            errors.append("assembly_rules must be a list")

        if "validation_rules" not in contract:
            errors.append("Missing required field: validation_rules")

        return errors

    @classmethod
    def _verify_v3_contract_fields(
        cls,
        contract: dict[str, Any],
        base_slot: str,
        class_registry: dict[str, type[object]] | None = None,
    ) -> list[str]:
        """Verify required fields for v3 contract format.

        Args:
            contract: Parsed contract dict
            base_slot: Base slot identifier
            class_registry: Optional class registry for method verification

        Returns:
            List of error messages (empty if all checks pass)
        """
        errors: list[str] = []

        if "identity" not in contract:
            errors.append("Missing required field: identity")
        else:
            identity = contract["identity"]
            if "base_slot" not in identity:
                errors.append("identity missing 'base_slot' field")
            elif identity["base_slot"] != base_slot:
                errors.append(
                    f"identity.base_slot mismatch: expected {base_slot}, got {identity['base_slot']}"
                )

        if "method_binding" not in contract:
            errors.append("Missing required field: method_binding")
        else:
            method_binding = contract["method_binding"]
            orchestration_mode = method_binding.get("orchestration_mode", "single_method")

            if orchestration_mode == "multi_method_pipeline":
                if "methods" not in method_binding:
                    errors.append("method_binding missing 'methods' array for multi_method_pipeline mode")
                elif not isinstance(method_binding["methods"], list):
                    errors.append("method_binding.methods must be a list")
                else:
                    methods = method_binding["methods"]
                    if not methods:
                        errors.append("method_binding.methods is empty")
                    else:
                        for idx, method_spec in enumerate(methods):
                            if not isinstance(method_spec, dict):
                                errors.append(f"methods[{idx}] is not a dict")
                                continue
                            if "class_name" not in method_spec:
                                errors.append(f"methods[{idx}] missing 'class_name' field")
                            if "method_name" not in method_spec:
                                errors.append(f"methods[{idx}] missing 'method_name' field")

                            if class_registry is not None and "class_name" in method_spec:
                                class_name = method_spec["class_name"]
                                if class_name not in class_registry:
                                    errors.append(
                                        f"methods[{idx}]: class '{class_name}' not found in class registry"
                                    )
            elif "class_name" not in method_binding and "primary_method" not in method_binding:
                errors.append(
                    "method_binding missing 'class_name' or 'primary_method' for single_method mode"
                )
            else:
                class_name = method_binding.get("class_name")
                if not class_name and "primary_method" in method_binding:
                    class_name = method_binding["primary_method"].get("class_name")

                if class_name and class_registry is not None:
                    if class_name not in class_registry:
                        errors.append(
                            f"method_binding: class '{class_name}' not found in class registry"
                        )

        if "evidence_assembly" not in contract:
            errors.append("Missing required field: evidence_assembly")
        else:
            evidence_assembly = contract["evidence_assembly"]
            if "assembly_rules" not in evidence_assembly:
                errors.append("evidence_assembly missing 'assembly_rules' field")
            elif not isinstance(evidence_assembly["assembly_rules"], list):
                errors.append("evidence_assembly.assembly_rules must be a list")

        if "validation_rules" not in contract:
            errors.append("Missing required field: validation_rules")
        if "question_context" not in contract:
            errors.append("Missing required field: question_context")
        else:
            question_context = contract["question_context"]
            if "expected_elements" not in question_context:
                errors.append("question_context missing 'expected_elements' field")

        if "error_handling" not in contract:
            errors.append("Missing required field: error_handling")

        return errors

    @classmethod
    def _get_schema_validator(cls, version: str = "v2") -> Draft7Validator:
        """Get schema validator for the specified contract version.

        Args:
            version: Contract version ("v2" or "v3")

        Returns:
            Draft7Validator for the specified version
        """
        if version not in cls._schema_validators:
            # Fallback for schema path (user reported misconfiguration)
            if version == "v3":
                schema_path = (
                    PROJECT_ROOT
                    / "config"
                    / "schemas"
                    / "executor_contract.v3.schema.json"
                )
            else:
                schema_path = PROJECT_ROOT / "config" / "executor_contract.schema.json"

            # If default path doesn't exist, try local path in Phase_two/json_files_phase_two
            if not schema_path.exists():
                local_path = (
                    PROJECT_ROOT
                    / "src"
                    / "canonic_phases"
                    / "Phase_two"
                    / "json_files_phase_two"
                    / f"executor_contract.{version}.schema.json"
                )
                if local_path.exists():
                    schema_path = local_path
                else:
                     # Attempt to construct minimal schema in memory if files missing
                     # to prevent crashing if schema assets are misplaced
                     import logging
                     logging.warning(f"Schema file missing at {schema_path} and {local_path}. Using minimal fallback.")
                     minimal_schema = {"type": "object", "additionalProperties": True}
                     cls._schema_validators[version] = Draft7Validator(minimal_schema)
                     return cls._schema_validators[version]

            if not schema_path.exists():
                raise FileNotFoundError(f"Contract schema not found: {schema_path}")
            schema = json.loads(schema_path.read_text(encoding="utf-8"))
            cls._schema_validators[version] = Draft7Validator(schema)
        return cls._schema_validators[version]

    @classmethod
    def _detect_contract_version(cls, contract: dict[str, Any]) -> str:
        """Detect contract version from structure.

        v3 contracts have: identity, executor_binding, method_binding, question_context
        v2 contracts have: method_inputs, assembly_rules at top level

        Returns:
            "v3" or "v2"
        """
        v3_indicators = [
            "identity",
            "executor_binding",
            "method_binding",
            "question_context",
        ]
        if all(key in contract for key in v3_indicators):
            return "v3"
        return "v2"

    @classmethod
    def _load_contract(cls, question_id: str | None = None) -> dict[str, Any]:
        base_slot = cls.get_base_slot()

        # Use specific question_id if provided, otherwise derive base Q-id from base_slot
        if question_id:
            cache_key = f"{base_slot}:{question_id}"
            q_id = question_id
        else:
            cache_key = base_slot
            dimension = int(base_slot[1])
            question = int(base_slot[4])
            q_number = (dimension - 1) * 5 + question
            q_id = f"Q{q_number:03d}"
        if cache_key in cls._contract_cache:
            return cls._contract_cache[cache_key]

        contracts_dir = PROJECT_ROOT / "src" / "farfan_pipeline" / "phases" / "Phase_two" / "json_files_phase_two" / "executor_
contracts"

        v3_path = contracts_dir / f"{base_slot}.v3.json"
        v2_path = contracts_dir / f"{base_slot}.json"
        v3_specialized_path = contracts_dir / "specialized" / f"{q_id}.v3.json"
        v2_specialized_path = contracts_dir / "specialized" / f"{q_id}.json"

        if v3_specialized_path.exists():
            contract_path = v3_specialized_path
            expected_version = "v3"
        elif v2_specialized_path.exists():
            contract_path = v2_specialized_path
            expected_version = "v2"
        elif v3_path.exists():
            contract_path = v3_path
            expected_version = "v3"
        elif v2_path.exists():
            contract_path = v2_path
            expected_version = "v2"
        else:
            raise FileNotFoundError(
                f"Contract not found for {base_slot} / {q_id}. "
                f"Tried: {v3_path}, {v2_path}, {v3_specialized_path}, {v2_specialized_path}"
            )

        contract = json.loads(contract_path.read_text(encoding="utf-8"))

        # Detect actual version from structure
        detected_version = cls._detect_contract_version(contract)
        if detected_version != expected_version:
            import logging

            logging.warning(
                f"Contract {contract_path.name} has structure of {detected_version} "
                f"but file naming suggests {expected_version}"
            )

        # Validate with appropriate schema
        validator = cls._get_schema_validator(detected_version)
        errors = sorted(validator.iter_errors(contract), key=lambda e: e.path)
        if errors:
            messages = "; ".join(err.message for err in errors)
            raise ValueError(
                f"Contract validation failed for {base_slot} ({detected_version}): {messages}"
            )

        # Tag contract with version for later use
        contract["_contract_version"] = detected_version

        contract_version = contract.get("contract_version")
        if contract_version and not str(contract_version).startswith("2"):
            raise ValueError(
                f"Unsupported contract_version {contract_version} for {base_slot}; expected v2.x"
            )

        identity_base_slot = contract.get("identity", {}).get("base_slot")
        if identity_base_slot and identity_base_slot != base_slot:
            raise ValueError(
                f"Contract base_slot mismatch: expected {base_slot}, found {identity_base_slot}"
            )

        cls._contract_cache[cache_key] = contract
        return contract

    def _validate_signal_requirements(
        self,
        signal_pack: Any,
        signal_requirements: dict[str, Any],
        base_slot: str,
    ) -> None:
        """Validate that signal requirements from contract are met.

        Args:
            signal_pack: Signal pack retrieved from registry (may be None)
            signal_requirements: signal_requirements section from contract
            base_slot: Base slot identifier for error messages

        Raises:
            RuntimeError: If mandatory signal requirements are not met
        """
        mandatory_signals = signal_requirements.get("mandatory_signals", [])
        minimum_threshold = signal_requirements.get("minimum_signal_threshold", 0.0)

        # Check if mandatory signals are required but no signal pack available
        if mandatory_signals and signal_pack is None:
            raise RuntimeError(
                f"Contract {base_slot} requires mandatory signals {mandatory_signals}, "
                "but no signal pack was retrieved from registry. "
                "Ensure signal registry is properly configured and policy_area_id is valid."
            )

        # If signal pack exists, validate signal strength
        if signal_pack is not None and minimum_threshold > 0:
            # Check if signal pack has strength attribute
            if hasattr(signal_pack, "strength") or (
                isinstance(signal_pack, dict) and "strength" in signal_pack
            ):
                strength = (
                    signal_pack.strength
                    if hasattr(signal_pack, "strength")
                    else signal_pack["strength"]
                )
                if strength < minimum_threshold:
                    raise RuntimeError(
                        f"Contract {base_slot} requires minimum signal threshold {minimum_threshold}, "
                        f"but signal pack has strength {strength}. "
                        "Signal quality is insufficient for execution."
                    )

    @staticmethod
    def _set_nested_value(
        target_dict: dict[str, Any], key_path: str, value: Any
    ) -> None:
        """Set a value in a nested dict using dot-notation key path.

        Args:
            target_dict: The dictionary to modify
            key_path: Dot-separated path (e.g., "text_mining.critical_links")
            value: The value to set

        Example:
            _set_nested_value(d, "a.b.c", 123) â\206\222 d["a"]["b"]["c"] = 123
        """
        keys = key_path.split(".")
        current = target_dict

        # Navigate to the parent of the final key, creating dicts as needed
        for key in keys[:-1]:
            if key not in current:
                current[key] = {}
            elif not isinstance(current[key], dict):
                # Key exists but is not a dict, cannot nest further
                raise ValueError(
                    f"Cannot set nested value at '{key_path}': "
                    f"intermediate key '{key}' exists but is not a dict"
                )
            current = current[key]

        # Set the final key
        current[keys[-1]] = value

    def _check_failure_contract(
        self, evidence: dict[str, Any], error_handling: dict[str, Any]
    ) -> None:
        failure_contract = error_handling.get("failure_contract", {})
        abort_conditions = failure_contract.get("abort_if", [])
        if not abort_conditions:
            return

        emit_code = failure_contract.get("emit_code", "GENERIC_ABORT")

        for condition in abort_conditions:
            # Example condition check. This could be made more sophisticated.
            if condition == "missing_required_element" and evidence.get(
                "validation", {}
            ).get("errors"):
                # This logic assumes errors from the validator imply a missing required element,
                # which is true with our new validator.
                raise ValueError(
                    f"Execution aborted by failure contract due to '{condition}'. Emit code: {emit_code}"
                )
            if condition == "incomplete_text" and not evidence.get("metadata", {}).get(
                "text_complete", True
            ):
                raise ValueError(
                    f"Execution aborted by failure contract due to '{condition}'. Emit code: {emit_code}"
                )

    @classmethod
    def load_all_contracts(
        cls,
        contracts_dir: str | None = None,
        version: str = "v3",
        validate_schema: bool = True,
    ) -> list[dict[str, Any]]:
        """Load all 300 specialized contracts from directory.

        Batch loads Q001.v3.json through Q300.v3.json with validation and caching.
        Leverages existing _load_contract() infrastructure for consistency.

        Args:
            contracts_dir: Directory containing specialized contracts.
                          Defaults to PROJECT_ROOT/../executor_contracts/specialized/
            version: Contract version to load ("v2" or "v3")
            validate_schema: Whether to validate contracts against JSON schema

        Returns:
            List of 300 contract dicts, ordered by question_id (Q001-Q300)
        Raises:
            FileNotFoundError: If contracts directory doesn’t exist
            ValueError: If any contract fails to load or validate

        Example:
            >>> contracts = BaseExecutorWithContract.load_all_contracts()
            >>> len(contracts)
            300
            >>> contracts[0][’identity’][’question_id’]
            ’Q001’
        """
        from pathlib import Path

        if contracts_dir is None:
            contracts_dir = str(PROJECT_ROOT / ".." / "executor_contracts" / "specialized")

        contracts_path = Path(contracts_dir)
        if not contracts_path.exists():
            raise FileNotFoundError(
                f"Contracts directory not found: {contracts_dir}"
            )

        contracts = []
        failed_loads = []

        for q_num in range(1, 301):
            question_id = f"Q{q_num:03d}"
            try:
                # Use existing _load_contract infrastructure
                contract = cls._load_contract_from_file(
                    question_id=question_id,
                    contracts_dir=contracts_dir,
                    version=version,
                    validate_schema=validate_schema
                )
                contracts.append(contract)
            except Exception as e:
                failed_loads.append(f"{question_id}: {str(e)}")

        if failed_loads:
            error_msg = (
                f"Failed to load {len(failed_loads)} contracts:\n"
                + "\n".join(failed_loads[:10])
            )
            if len(failed_loads) > 10:
                error_msg += f"\n... and {len(failed_loads) - 10} more"
            raise ValueError(error_msg)

        return contracts

    @classmethod
    def _load_contract_from_file(
        cls,
        question_id: str,
        contracts_dir: str,
        version: str = "v3",
        validate_schema: bool = True
    ) -> dict[str, Any]:
        """Load a single contract from file with caching and validation.

        Helper method for load_all_contracts() that handles individual contract loading.
        Integrates with existing _contract_cache infrastructure.

        Args:
            question_id: Question identifier (e.g., "Q001")
            contracts_dir: Directory containing contract files
            version: Contract version ("v2" or "v3")
            validate_schema: Whether to validate against JSON schema

        Returns:
            Contract dictionary

        Raises:
            FileNotFoundError: If contract file doesn’t exist
            json.JSONDecodeError: If contract JSON is invalid
            ValueError: If contract fails schema validation
        """
        from pathlib import Path

        cache_key = f"{question_id}_{version}_{contracts_dir}"

        if cache_key in cls._contract_cache:
            return cls._contract_cache[cache_key]

        contracts_path = Path(contracts_dir)
        ext = f".{version}.json" if version == "v3" else ".json"
        contract_file = contracts_path / f"{question_id}{ext}"

        if not contract_file.exists():
            raise FileNotFoundError(
                f"Contract file not found: {contract_file}"
            )

        with open(contract_file, "r", encoding="utf-8") as f:
            contract = json.load(f)
        if validate_schema and version == "v3":
            cls._validate_contract_schema(contract, question_id)

        cls._contract_cache[cache_key] = contract
        return contract

    @classmethod
    def _validate_contract_schema(
        cls,
        contract: dict[str, Any],
        question_id: str
    ) -> None:
        """Validate contract against v3 JSON schema.

        Args:
            contract: Contract dictionary to validate
            question_id: Question identifier for error messages

        Raises:
            ValueError: If contract fails schema validation
        """
        required_v3_keys = [
            "identity",
            "executor_binding",
            "method_binding",
            "question_context",
            "evidence_assembly",
            "output_contract"
        ]

        missing_keys = [key for key in required_v3_keys if key not in contract]
        if missing_keys:
            raise ValueError(
                f"Contract {question_id} missing required v3 keys: {missing_keys}"
            )

        identity = contract.get("identity", {})
        if identity.get("question_id") != question_id:
            raise ValueError(
                f"Contract identity mismatch: expected {question_id}, "
                f"got {identity.get('question_id')}"
            )

    @classmethod
    def clear_contract_cache(cls) -> None:
        """Clear the contract cache.

        Useful for testing or when contracts are updated on disk.
        """
        cls._contract_cache.clear()

    @classmethod
    def get_cached_contract_count(cls) -> int:
        """Get number of contracts currently in cache.

        Returns:
            Number of cached contracts
        """
        return len(cls._contract_cache)

    def execute(
        self,
        document: PreprocessedDocument,
        method_executor: MethodExecutor,
        *,
        question_context: dict[str, Any],
    ) -> dict[str, Any]:
        if method_executor is not self.method_executor:
            raise RuntimeError(
                "Mismatched MethodExecutor instance for contract executor"
            )

        base_slot = self.get_base_slot()
        if question_context.get("base_slot") != base_slot:
            raise ValueError(
                f"Question base_slot {question_context.get('base_slot')} does not match executor {base_slot}"
            )

        question_id = question_context.get("question_id")
        contract = self._load_contract(question_id=question_id)
        contract_version = contract.get("_contract_version", "v2")

        if contract_version == "v3":
            return self._execute_v3(document, question_context, contract)
        else:
            return self._execute_v2(document, question_context, contract)

    def _execute_v2(
        self,
        document: PreprocessedDocument,
        question_context: dict[str, Any],
        contract: dict[str, Any],
    ) -> dict[str, Any]:
        """Execute using v2 contract format (legacy)."""
        base_slot = self.get_base_slot()
        question_id = question_context.get("question_id")
        question_global = question_context.get("question_global")
        policy_area_id = question_context.get("policy_area_id")
        identity = question_context.get("identity", {})
        patterns = question_context.get("patterns", [])
        expected_elements = question_context.get("expected_elements", [])

        # JOBFRONT 3: Use enriched signal packs if available
        signal_pack = None
        enriched_pack = None
        applicable_patterns = patterns  # Default to contract patterns
        document_context = {}

        if self._use_enriched_signals and policy_area_id in self.enriched_packs:
            # Use enriched intelligence layer
            enriched_pack = self.enriched_packs[policy_area_id]
            signal_pack = enriched_pack.base_pack  # Maintain compatibility

            # Create document context from available metadata
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_intelligence_layer import (
                create_document_context,
            )

            doc_metadata = getattr(document, "metadata", {})
            document_context = create_document_context(
                section=doc_metadata.get("section"),
                chapter=doc_metadata.get("chapter"),
                page=doc_metadata.get("page"),
                policy_area=policy_area_id
            )

            # Get context-filtered patterns (REFACTORING #6: context scoping)
            applicable_patterns = enriched_pack.get_patterns_for_context(document_context)

            # Expand patterns semantically (REFACTORING #2: semantic expansion)
            if applicable_patterns and isinstance(applicable_patterns[0], dict):
                pattern_strings = [p.get('pattern', p) if isinstance(p, dict) else p for p in applicable_patterns]
            else:
                pattern_strings = applicable_patterns

            expanded_patterns = enriched_pack.expand_patterns(pattern_strings)
            applicable_patterns = expanded_patterns

        elif self.signal_registry is not None and hasattr(self.signal_registry, "get") and policy_area_id:
            # Fallback to legacy signal registry
            signal_pack = self.signal_registry.get(policy_area_id)

        common_kwargs: dict[str, Any] = {
            "document": document,
            "base_slot": base_slot,
            "raw_text": getattr(document, "raw_text", None),
            "text": getattr(document, "raw_text", None),
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": identity.get("dimension_id"),
            "cluster_id": identity.get("cluster_id"),
            "signal_pack": signal_pack,
            "enriched_pack": enriched_pack,  # NEW: Pass enriched pack
            "document_context": document_context,  # NEW: Pass document context
            "question_patterns": applicable_patterns,  # Use filtered/expanded patterns
            "expected_elements": expected_elements,
        }

        method_outputs: dict[str, Any] = {}
        method_inputs = contract.get("method_inputs", [])
        indexed = list(enumerate(method_inputs))
        sorted_inputs = sorted(
            indexed, key=lambda pair: (pair[1].get("priority", 2), pair[0])
        )

        calibration_results = {}
        calibration_weights = {}

        for _, entry in sorted_inputs:
            class_name = entry["class"]
            method_name = entry["method"]
            provides = entry.get("provides", [])
            extra_args = entry.get("args", {})

            payload = {**common_kwargs, **extra_args}

            method_id = f"{class_name}.{method_name}"
            calibration_score = None

            if self.calibration_orchestrator:
                try:
                    from cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator import (
                        MethodBelowThresholdError,
                    )

                    calibration_result = self.calibration_orchestrator.calibrate(
                        method_id=method_id,
                        context=payload,
                        evidence=None
                    )
                    calibration_score = calibration_result.final_score
                    calibration_results[method_id] = calibration_result.to_dict()

                    import logging
                    logger = logging.getLogger(__name__)
                    logger.info(
                        f"[{base_slot}] Calibration: {method_id} â\206\222 {calibration_score:.3f}"
                    )

                except MethodBelowThresholdError as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    calibration_score = e.score

                    should_execute, reason = self.calibration_policy.should_execute_method(
                        method_id, calibration_score
                    )

                    if not should_execute:
                        logger.error(
                            f"[{base_slot}] Method {method_id} SKIPPED: {reason}"
                        )
                        continue
                    else:
                        logger.warning(
                            f"[{base_slot}] Method {method_id} below threshold but executing: {reason}"
                        )
                except Exception as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    logger.warning(f"[{base_slot}] Calibration error for {method_id}: {e}")

            should_execute, exec_reason = self.calibration_policy.should_execute_method(
                method_id, calibration_score
            )

            if not should_execute:
                import logging
                logger = logging.getLogger(__name__)
                logger.warning(f"[{base_slot}] Skipping {method_id}: {exec_reason}")
                continue

            base_weight = entry.get("weight", 1.0)
            weight_info = self.calibration_policy.compute_adjusted_weight(
                base_weight=base_weight,
                calibration_score=calibration_score,
                method_id=method_id,
            )
            calibration_weights[method_id] = weight_info.to_dict()

            self.calibration_policy.record_influence(
                phase_id=2,
                method_id=method_id,
                calibration_score=calibration_score or 0.0,
                weight_adjustment=base_weight - weight_info.adjusted_weight,
                influenced_output=weight_info.adjusted_weight != base_weight,
                base_slot=base_slot,
                question_id=question_id,
            )

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **payload,
            )

            if "_calibration_weight" not in result or not isinstance(result, dict):
                if isinstance(result, dict):
                    result["_calibration_weight"] = weight_info.adjusted_weight
                    result["_calibration_score"] = calibration_score
                    result["_calibration_quality_band"] = weight_info.quality_band

            if "signal_pack" in payload and payload["signal_pack"] is not None:
                if "_signal_usage" not in method_outputs:
                    method_outputs["_signal_usage"] = []
                method_outputs["_signal_usage"].append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": payload["signal_pack"].policy_area,
                        "version": payload["signal_pack"].version,
                    }
                )

            if isinstance(provides, str):
                method_outputs[provides] = result
            else:
                for key in provides:
                    method_outputs[key] = result

        assembly_rules = contract.get("assembly_rules", [])

        # NEW: Use EvidenceNexus instead of legacy EvidenceAssembler
        nexus_result = process_evidence(
            method_outputs=method_outputs,
            assembly_rules=assembly_rules,
            validation_rules=contract.get("validation_rules", []),
            question_context={
                "question_id": question_id,
                "question_global": question_global,
                "expected_elements": expected_elements,
                "patterns": applicable_patterns,
                # Provide raw text so EvidenceNexus can run pattern extraction deterministically.
                "raw_text": getattr(document, "raw_text", "") or "",
            },
            signal_pack=signal_pack,  # SISAS: Enable signal provenance
            contract=contract,
        )

        evidence = nexus_result["evidence"]
        trace = nexus_result["trace"]
        validation = nexus_result["validation"]

        # JOBFRONT 3: Extract structured evidence if enriched pack available
        completeness = 1.0
        missing_elements = []
        patterns_used = []

        if enriched_pack is not None and expected_elements:
            # Build signal node for evidence extraction
            signal_node = {
                "id": question_id,
                "expected_elements": expected_elements,
                "patterns": applicable_patterns,
                "validations": contract.get("validation_rules", [])
            }

            # Extract structured evidence (REFACTORING #5: evidence structure)
            evidence_result = enriched_pack.extract_evidence(
                text=getattr(document, "raw_text", ""),
                signal_node=signal_node,
                document_context=document_context
            )

            # Merge structured evidence into result
            for element_type, matches in evidence_result.evidence.items():
                if element_type not in evidence:
                    evidence[element_type] = matches

            completeness = evidence_result.completeness
            missing_elements = evidence_result.missing_required

            # Track patterns used (for confidence calculation)
            if isinstance(applicable_patterns, list):
                patterns_used = [p.get('id', p) if isinstance(p, dict) else p
                                 for p in applicable_patterns[:10]]  # Top 10

        # Note: Validation is now handled by EvidenceNexus above
        error_handling = contract.get("error_handling", {})

        # JOBFRONT 3: Add contract validation if enriched pack available
        contract_validation = None
        if enriched_pack is not None:
            # Build signal node for contract validation
            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules,
                "expected_elements": expected_elements
            }

            # Validate with contracts (REFACTORING #4: contract validation)
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator if self._use_validation_orchestrator else None,
                auto_register=self._use_validation_orchestrator
            )

            # Merge contract validation into standard validation
            if not contract_validation.passed:
                validation["status"] = "failed"
                validation["errors"] = validation.get("errors", [])
                validation["errors"].append({
                    "error_code": contract_validation.error_code,
                    "condition_violated": contract_validation.condition_violated,
                    "remediation": contract_validation.remediation,
                    "failures_detailed": [
                        {
                            "type": f.failure_type,
                            "field": f.field_name,
                            "message": f.message,
                            "severity": f.severity,
                            "remediation": f.remediation
                        }
                        for f in contract_validation.failures_detailed[:5]
                    ]
                })
                validation["contract_failed"] = True
                validation["contract_validation_details"] = {
                    "error_code": contract_validation.error_code,
                    "diagnostics": contract_validation.diagnostics,
                    "total_failures": len(contract_validation.failures_detailed)
                }
        elif self._use_validation_orchestrator:
            # Even without enriched pack, use validation orchestrator with basic validation
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": {"rules": validation_rules},
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator,
                auto_register=True
            )
        if error_handling:
            evidence_with_validation = {**evidence, "validation": validation}
            self._check_failure_contract(evidence_with_validation, error_handling)

        human_answer_template = contract.get("human_answer_template", "")
        human_answer = ""
        if human_answer_template:
            try:
                human_answer = human_answer_template.format(**evidence)
            except KeyError as e:
                human_answer = f"Error formatting human answer: Missing key {e}. Template: '{human_answer_template}'"
                import logging

                logging.warning(human_answer)

        result = {
            "base_slot": base_slot,
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": identity.get("dimension_id"),
            "cluster_id": identity.get("cluster_id"),
            "evidence": evidence,
            "validation": validation,
            "trace": trace,
            "human_answer": human_answer,
            # JOBFRONT 3: Add intelligence layer metadata
            "completeness": completeness,
            "missing_elements": missing_elements,
            "patterns_used": patterns_used,
            "enriched_signals_enabled": enriched_pack is not None,
            # VALIDATION ORCHESTRATOR: Add validation tracking metadata
            "contract_validation": {
                "enabled": contract_validation is not None,
                "passed": contract_validation.passed if contract_validation else None,
                "error_code": contract_validation.error_code if contract_validation else None,
                "failure_count": len(contract_validation.failures_detailed) if contract_validation else 0,
                "orchestrator_registered": self._use_validation_orchestrator
            },
            # CALIBRATION: Add calibration metadata
            "calibration_metadata": {
                "enabled": self.calibration_orchestrator is not None,
                "results": calibration_results,
                "weights": calibration_weights,
                "summary": {
                    "total_methods": len(calibration_results),
                    "average_score": sum(
                        cr["final_score"] for cr in calibration_results.values()
                    ) / len(calibration_results) if calibration_results else 0.0,
                    "min_score": min(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                    "max_score": max(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                    "methods_executed": len(method_outputs),
                    "methods_skipped": len(method_inputs) - len(method_outputs),
                }
            }
        }

        return result

    def _execute_v3(
        self,
        document: PreprocessedDocument,
        question_context_external: dict[str, Any],
        contract: dict[str, Any],
    ) -> dict[str, Any]:
```

Phase 2 Source Code

```
        """Execute using v3 contract format.

        In v3, contract contains all context, so we use contract['question_context']
        instead of question_context_external (which comes from orchestrator).
        """
        # Extract identity from contract
        identity = contract["identity"]
        base_slot = identity["base_slot"]
        question_id = identity["question_id"]
        dimension_id = identity["dimension_id"]
        policy_area_id = identity["policy_area_id"]

        # CALIBRATION ENFORCEMENT: Verify calibration status before execution
        calibration = contract.get("calibration", {})
        calibration_status = calibration.get("status", "placeholder")
        if calibration_status == "placeholder":
            import logging
            logging.info(
                f"Contract {base_slot} has placeholder calibration. "
                "Injecting live calibration parameters from UnitOfAnalysisLoader..."
            )
            # Override status to enable execution with alive parameters
            calibration["status"] = "calibrated_alive"
            calibration["note"] = "Live parameters injected from canonic_description_unit_analysis.json"

        # Extract question context from contract (source of truth for v3)
        question_context = contract["question_context"]
        question_global = question_context_external.get(
            "question_global"
        )  # May come from orchestrator
        patterns = question_context.get("patterns", [])
        expected_elements = question_context.get("expected_elements", [])

        # Signal pack
        signal_pack = None
        if (
            self.signal_registry is not None
            and hasattr(self.signal_registry, "get")
            and policy_area_id
        ):
            signal_pack = self.signal_registry.get(policy_area_id)

        # SISAS: Inject consumption tracking (utility + proof chain)
        consumption_tracker = None
        try:
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_consumption_integration import (
                inject_consumption_tracking,
            )

            consumption_tracker = inject_consumption_tracking(
                executor=self,
                question_id=question_id,
                policy_area_id=policy_area_id,
                # Deterministic: do not depend on wall clock time for proofs.
                injection_time=0.0,
            )
        except Exception:
            consumption_tracker = None

        # Build document context (for scope coherence + context-aware pattern filtering)
        document_context: dict[str, Any] = {}
        try:
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_context_scoper import (
                create_document_context,
            )

            doc_metadata = getattr(document, "metadata", {}) or {}
            document_context = create_document_context(
                section=doc_metadata.get("section"),
                chapter=doc_metadata.get("chapter"),
                page=doc_metadata.get("page"),
                policy_area=policy_area_id,
            )
        except Exception:
            document_context = {"policy_area": policy_area_id}

        # SIGNAL REQUIREMENTS VALIDATION: Verify signal requirements from contract
        signal_requirements = contract.get("signal_requirements", {})
        if signal_requirements:
            self._validate_signal_requirements(
                signal_pack, signal_requirements, base_slot
            )

        # Extract method binding
        method_binding = contract["method_binding"]
        orchestration_mode = method_binding.get("orchestration_mode", "single_method")

        # Prepare common kwargs
        common_kwargs: dict[str, Any] = {
            "document": document,
            "base_slot": base_slot,
            "raw_text": getattr(document, "raw_text", None),
            "text": getattr(document, "raw_text", None),
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": dimension_id,
            "cluster_id": identity.get("cluster_id"),
            "signal_pack": signal_pack,
            "question_patterns": patterns,
            "expected_elements": expected_elements,
            "question_context": question_context,
        }

        # Execute methods based on orchestration mode
        method_outputs: dict[str, Any] = {}
        signal_usage_list: list[dict[str, Any]] = []
        calibration_results: dict[str, Any] = {}

        if orchestration_mode == "multi_method_pipeline":
            # Multi-method execution: process all methods in priority order
            methods = method_binding.get("methods", [])
            if not methods:
                raise ValueError(
                    f"orchestration_mode is ’multi_method_pipeline’ but no methods array found in method_binding for {base_slot
}"
                )

            # Sort by priority (lower priority number = execute first)
            sorted_methods = sorted(methods, key=lambda m: m.get("priority", 99))

            for method_spec in sorted_methods:
                class_name = method_spec["class_name"]
                method_name = method_spec["method_name"]
                provides = method_spec.get("provides", f"{class_name}.{method_name}")
                priority = method_spec.get("priority", 99)

                method_id = f"{class_name}.{method_name}"

                if self.calibration_orchestrator:
                    try:
                        from cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator import (
                            MethodBelowThresholdError,
                        )

                        calibration_result = self.calibration_orchestrator.calibrate(
                            method_id=method_id,
                            context=common_kwargs,
                            evidence=None
                        )

                        calibration_results[method_id] = calibration_result.to_dict()

                        import logging
                        logger = logging.getLogger(__name__)
                        logger.info(
                            f"[{base_slot}] Calibration: {method_id} â\206\222 {calibration_result.final_score:.3f}"
                        )

                    except MethodBelowThresholdError as e:
                        import logging
                        logger = logging.getLogger(__name__)
                        logger.error(
                            f"[{base_slot}] Method {method_id} FAILED calibration: "
                            f"score={e.score:.3f}, threshold={e.threshold:.3f}"
                        )
                        raise RuntimeError(
                            f"Method {method_id} failed calibration threshold"
                        ) from e
                    except Exception as e:
                        import logging
                        logger = logging.getLogger(__name__)
                        logger.warning(f"[{base_slot}] Calibration error for {method_id}: {e}")

                try:
                    result = self.method_executor.execute(
                        class_name=class_name,
                        method_name=method_name,
                        **common_kwargs,
                    )

                    # Store result using nested key structure (e.g., "text_mining.critical_links")
                    self._set_nested_value(method_outputs, provides, result)

                    # Track signal usage for this method
                    if signal_pack is not None:
                        signal_usage_list.append(
                            {
                                "method": f"{class_name}.{method_name}",
                                "policy_area": signal_pack.policy_area,
                                "version": signal_pack.version,
                                "priority": priority,
                            }
                        )

                except Exception as exc:
                    import logging

                    logging.error(
                        f"Method execution failed in multi-method pipeline: {class_name}.{method_name}",
                        exc_info=True,
                    )
                    # Store error in trace for debugging
                    # Store error in a flat structure under _errors[provides]
                    if "_errors" not in method_outputs or not isinstance(
                        method_outputs["_errors"], dict
                    ):
                        method_outputs["_errors"] = {}
                    method_outputs["_errors"][provides] = {
                        "error": str(exc),
                        "method": f"{class_name}.{method_name}",
                    }
                    # Re-raise if error_handling policy requires it
                    error_handling = contract.get("error_handling", {})
                    on_method_failure = error_handling.get(
                        "on_method_failure", "propagate_with_trace"
                    )
                    if on_method_failure == "raise":
                        raise
                    # Otherwise continue with other methods

        else:
            # Single-method execution (backward compatible, default)
            class_name = method_binding.get("class_name")
            method_name = method_binding.get("method_name")

            if not class_name or not method_name:
                # Try primary_method if direct class_name/method_name not found
                primary_method = method_binding.get("primary_method", {})
                class_name = primary_method.get("class_name") or class_name
                method_name = primary_method.get("method_name") or method_name

            if not class_name or not method_name:
                raise ValueError(
                    f"Invalid method_binding for {base_slot}: missing class_name or method_name"
                )

            method_id = f"{class_name}.{method_name}"

            if self.calibration_orchestrator:
                try:
                    from cross_cutting_infrastructure.capaz_calibration_parmetrization.calibration_orchestrator import (
                        MethodBelowThresholdError,
                    )

                    calibration_result = self.calibration_orchestrator.calibrate(
                        method_id=method_id,
                        context=common_kwargs,
                        evidence=None
                    )

                    calibration_results[method_id] = calibration_result.to_dict()

                    import logging
                    logger = logging.getLogger(__name__)
                    logger.info(
                        f"[{base_slot}] Calibration: {method_id} â\206\222 {calibration_result.final_score:.3f}"
                    )

                except MethodBelowThresholdError as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    logger.error(
                        f"[{base_slot}] Method {method_id} FAILED calibration: "
                        f"score={e.score:.3f}, threshold={e.threshold:.3f}"
                    )
                    raise RuntimeError(
                        f"Method {method_id} failed calibration threshold"
                    ) from e
                except Exception as e:
                    import logging
                    logger = logging.getLogger(__name__)
                    logger.warning(f"[{base_slot}] Calibration error for {method_id}: {e}")

            result = self.method_executor.execute(
                class_name=class_name,
                method_name=method_name,
                **common_kwargs,
            )
            method_outputs["primary_analysis"] = result

            # Track signal usage
            if signal_pack is not None:
                signal_usage_list.append(
                    {
                        "method": f"{class_name}.{method_name}",
                        "policy_area": signal_pack.policy_area,
                        "version": signal_pack.version,
                    }
                )

        # Store signal usage in method_outputs for trace
        if signal_usage_list:
            method_outputs["_signal_usage"] = signal_usage_list

        # NEW: Evidence assembly and validation using EvidenceNexus
        # Note: EvidenceNexus extracts assembly_rules and validation_rules from contract directly
        validation_rules_section = contract.get("validation_rules", {})
        nexus_result = process_evidence(
            method_outputs=method_outputs,
            question_context={
                "question_id": question_id,
                "question_global": question_global,
                "policy_area_id": policy_area_id,
                "dimension_id": dimension_id,
                "expected_elements": expected_elements,
                "patterns": patterns,
                # Provide raw text so EvidenceNexus can run pattern extraction deterministically.
                "raw_text": getattr(document, "raw_text", "") or "",
                # Provide document context for scope coherence + context filters.
                "document_context": document_context,
                # Provide SISAS consumption tracker for proof + utilization metrics.
                "consumption_tracker": consumption_tracker,
            },
            signal_pack=signal_pack,  # SISAS: Enable signal provenance
            contract=contract,
        )

        evidence = nexus_result["evidence"]
        trace = nexus_result["trace"]
        validation = nexus_result["validation"]

        # Get error_handling for subsequent validation orchestrator
        error_handling = contract.get("error_handling", {})

        # Reconstruct validation_rules_object for compatibility with ValidationOrchestrator
        validation_rules = validation_rules_section.get("rules", [])
        na_policy = validation_rules_section.get("na_policy", "abort_on_critical")
        validation_rules_object = {"rules": validation_rules, "na_policy": na_policy}

        # CONTRACT VALIDATION with ValidationOrchestrator
        contract_validation = None
        if self._use_validation_orchestrator:
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_contract_validator import (
                validate_result_with_orchestrator,
            )

            signal_node_for_validation = {
                "id": question_id,
                "failure_contract": error_handling.get("failure_contract", {}),
                "validations": validation_rules_object,
                "expected_elements": expected_elements
            }

            contract_validation = validate_result_with_orchestrator(
                result=evidence,
                signal_node=signal_node_for_validation,
                orchestrator=self.validation_orchestrator,
                auto_register=True
            )

            # Merge contract validation failures into standard validation
            if not contract_validation.passed:
                validation["contract_validation_failed"] = True
                validation["contract_error_code"] = contract_validation.error_code
                validation["contract_remediation"] = contract_validation.remediation
                validation["contract_failures"] = [
                    {
                        "type": f.failure_type,
                        "field": f.field_name,
                        "message": f.message,
                        "severity": f.severity
                    }
                    for f in contract_validation.failures_detailed[:10]
                ]

        # Handle validation failures based on NA policy
        validation_passed = validation.get("passed", True)
        if not validation_passed:
            if na_policy == "abort_on_critical":
                # Error handling will check failure contract below
                pass  # Let error_handling section handle abort
            elif na_policy == "score_zero":
                # Mark result as failed with score zero
                validation["score"] = 0.0
                validation["quality_level"] = "FAILED_VALIDATION"
                validation["na_policy_applied"] = "score_zero"
            elif na_policy == "propagate":
                # Continue with validation errors in result
                validation["na_policy_applied"] = "propagate"
                validation["validation_failed"] = True

        # Error handling
        error_handling = contract["error_handling"]
        if error_handling:
            evidence_with_validation = {**evidence, "validation": validation}
            self._check_failure_contract(evidence_with_validation, error_handling)

        # Build result
        result_data = {
            "base_slot": base_slot,
            "question_id": question_id,
            "question_global": question_global,
            "policy_area_id": policy_area_id,
            "dimension_id": dimension_id,
            "cluster_id": identity.get("cluster_id"),
            "evidence": evidence,
            "validation": validation,
            "trace": trace,
            # CONTRACT VALIDATION METADATA
            "contract_validation": {
                "enabled": contract_validation is not None,
                "passed": contract_validation.passed if contract_validation else None,
                "error_code": contract_validation.error_code if contract_validation else None,
                "failure_count": len(contract_validation.failures_detailed) if contract_validation else 0,
                "orchestrator_registered": self._use_validation_orchestrator
            },
            # CALIBRATION METADATA
            "calibration_metadata": {
                "enabled": self.calibration_orchestrator is not None,
                "results": calibration_results,
                "summary": {
                    "total_methods": len(calibration_results),
                    "average_score": sum(
                        cr["final_score"] for cr in calibration_results.values()
                    ) / len(calibration_results) if calibration_results else 0.0,
                    "min_score": min(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                    "max_score": max(
                        (cr["final_score"] for cr in calibration_results.values()),
                        default=0.0
                    ),
                }
            }
        }

        # NEW: Record evidence provenance in EvidenceNexus internal store (if available in nexus_result)
        if "provenance_record" in nexus_result:
            # Provenance is now handled internally by EvidenceNexus
            result_data["provenance"] = nexus_result["provenance_record"]

        # NEW: Add EvidenceNexus scoring fields for Phase 3
        # These are essential for Phase 3 to extract scores for aggregation
        if "overall_confidence" in nexus_result:
            result_data["overall_confidence"] = nexus_result["overall_confidence"]
        if "completeness" in nexus_result:
            result_data["completeness"] = nexus_result["completeness"]
        if "calibrated_interval" in nexus_result:
            result_data["calibrated_interval"] = nexus_result["calibrated_interval"]
        if "synthesized_answer" in nexus_result:
            result_data["synthesized_answer"] = nexus_result["synthesized_answer"]

        # Validate output against output_contract schema if present
        output_contract = contract.get("output_contract", {})
        if output_contract and "schema" in output_contract:
            self._validate_output_contract(
                result_data, output_contract["schema"], base_slot
            )

        # NEW: Generate doctoral-level narrative using Carver synthesizer
        human_readable_config = output_contract.get("human_readable_output", {})
        if human_readable_config or nexus_result.get("graph"):
            # Use Carver to generate PhD-level narrative from evidence graph
            carver = DoctoralCarverSynthesizer()

            try:
                # Use synthesize_structured for full object access
                carver_answer = carver.synthesize_structured(evidence, contract)
                result_data["human_readable_output"] = carver_answer.to_human_readable()
                if hasattr(carver_answer, "synthesis_trace"):
                    result_data["carver_metrics"] = carver_answer.synthesis_trace
            except Exception as e:
                import logging
                logging.error(f"Carver synthesis failed: {e}", exc_info=True)
                # Fallback to basic template if Carver fails
                result_data["human_readable_output"] = self._generate_human_readable_output(
                    evidence, validation, human_readable_config, contract
                )
                result_data["carver_error"] = str(e)

        return result_data

    def _validate_output_contract(
        self, result: dict[str, Any], schema: dict[str, Any], base_slot: str
    ) -> None:
        """Validate result against output_contract schema with detailed error messages.

        Args:
            result: Result data to validate
            schema: JSON Schema from contract
            base_slot: Base slot identifier for error messages

        Raises:
            ValueError: If validation fails with detailed path information
        """
        from jsonschema import ValidationError, validate
        try:
            validate(instance=result, schema=schema)
        except ValidationError as e:
            # Enhanced error message with JSON path
            path = (
                ".".join(str(p) for p in e.absolute_path) if e.absolute_path else "root"
            )
            raise ValueError(
                f"Output contract validation failed for {base_slot} at '{path}': {e.message}. "
                f"Schema constraint: {e.schema}"
            ) from e

    def _generate_human_readable_output(
        self,
        evidence: dict[str, Any],
        validation: dict[str, Any],
        config: dict[str, Any],
        contract: dict[str, Any],
    ) -> str:
        """Generate production-grade human-readable output from template.

        Implements full template engine with:
        - Variable substitution with dot-notation: {evidence.elements_found_count}
        - Derived metrics: Automatic calculation of means, counts, percentages
        - List formatting: Convert arrays to markdown/html/plain_text lists
        - Methodological depth rendering: Full epistemological documentation
        - Multi-format support: markdown, html, plain_text with proper formatting

        Args:
            evidence: Evidence dict from executor
            validation: Validation dict
            config: human_readable_output config from contract
            contract: Full contract for methodological_depth access

        Returns:
            Formatted string in specified format
        """
        template_config = config.get("template", {})
        format_type = config.get("format", "markdown")
        methodological_depth_config = config.get("methodological_depth", {})

        # Build context for variable substitution
        context = self._build_template_context(evidence, validation, contract)

        # Render each template section
        sections = []

        # Title
        if "title" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["title"], context, format_type
                )
            )

        # Summary
        if "summary" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["summary"], context, format_type
                )
            )

        # Score section
        if "score_section" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["score_section"], context, format_type
                )
            )

        # Elements section
        if "elements_section" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["elements_section"], context, format_type
                )
            )

        # Details (list of items)
        if "details" in template_config and isinstance(
            template_config["details"], list
        ):
            detail_items = [
                self._render_template_string(item, context, format_type)
                for item in template_config["details"]
            ]
            sections.append(self._format_list(detail_items, format_type))

        # Interpretation
        if "interpretation" in template_config:
            # Add methodological interpretation if available
            context["methodological_interpretation"] = (
                self._render_methodological_depth(
                    methodological_depth_config, evidence, validation, format_type
                )
            )
            sections.append(
                self._render_template_string(
                    template_config["interpretation"], context, format_type
                )
            )

        # Recommendations
        if "recommendations" in template_config:
            sections.append(
                self._render_template_string(
                    template_config["recommendations"], context, format_type
                )
            )

        # Join sections with appropriate separator for format
        separator = (
            "\n\n"
            if format_type == "markdown"
            else "\n\n" if format_type == "plain_text" else "<br><br>"
        )
        return separator.join(filter(None, sections))

    def _build_template_context(
        self,
        evidence: dict[str, Any],
        validation: dict[str, Any],
        contract: dict[str, Any],
    ) -> dict[str, Any]:
        """Build comprehensive context for template variable substitution.

        Args:
            evidence: Evidence dict
            validation: Validation dict
            contract: Full contract

        Returns:
            Context dict with all variables and derived metrics
        """
        # Base context
        context = {
            "evidence": evidence.copy(),
            "validation": validation.copy(),
        }

        # Add derived metrics from evidence
        if "elements" in evidence and isinstance(evidence["elements"], list):
            context["evidence"]["elements_found_count"] = len(evidence["elements"])
            context["evidence"]["elements_found_list"] = self._format_evidence_list(
                evidence["elements"]
            )

        if "confidences" in evidence and isinstance(evidence["confidences"], list):
            confidences = evidence["confidences"]
            if confidences:
                context["evidence"]["confidence_scores"] = {
                    "mean": sum(confidences) / len(confidences),
                    "min": min(confidences),
                    "max": max(confidences),
                }

        if "patterns" in evidence and isinstance(evidence["patterns"], dict):
            context["evidence"]["pattern_matches_count"] = len(evidence["patterns"])

        # Add defaults for missing keys to prevent KeyError
        context["evidence"].setdefault("missing_required_elements", "None")
        context["evidence"].setdefault("official_sources_count", 0)
        context["evidence"].setdefault("quantitative_indicators_count", 0)
        context["evidence"].setdefault("temporal_series_count", 0)
        context["evidence"].setdefault("territorial_coverage", "Not specified")
        context["evidence"].setdefault(
            "recommendations", "No specific recommendations available"
        )

        # Add score and quality from validation or defaults
        context["score"] = validation.get("score", 0.0)
        context["quality_level"] = self._determine_quality_level(
            validation.get("score", 0.0)
        )

        return context

    def _determine_quality_level(self, score: float) -> str:
        """Determine quality level from score.

        Args:
            score: Numeric score (typically 0.0-3.0)

        Returns:
            Quality level string
        """
        if score >= 2.5:
            return "EXCELLENT"
        elif score >= 2.0:
            return "GOOD"
        elif score >= 1.0:
```

Phase 2 Source Code

```
            return "ACCEPTABLE"
        elif score > 0:
            return "INSUFFICIENT"
        else:
            return "FAILED"

    def _render_template_string(
        self, template: str, context: dict[str, Any], format_type: str
    ) -> str:
        """Render a template string with variable substitution.

        Supports dot-notation: {evidence.elements_found_count}
        Supports arithmetic: {score}/3.0 (rendered as-is, user interprets)

        Args:
            template: Template string with {variable} placeholders
            context: Context dict
            format_type: Output format (markdown, html, plain_text)

        Returns:
            Rendered string with variables substituted
        """
        import re

        def replace_var(match):
            var_path = match.group(1)
            try:
                # Handle dot-notation traversal
                keys = var_path.split(".")
                value = context
                for key in keys:
                    if isinstance(value, dict):
                        value = value[key]
                    else:
                        # Try to get attribute (for objects)
                        value = getattr(value, key, None)
                        if value is None:
                            return f"{{MISSING:{var_path}}}"

                # Format value appropriately
                if isinstance(value, float):
                    return f"{value:.2f}"
                elif isinstance(value, list | dict):
                    return str(value)  # Simple representation
                else:
                    return str(value)
            except (KeyError, AttributeError, TypeError):
                return f"{{MISSING:{var_path}}}"

        # Replace all {variable} patterns
        rendered = re.sub(r"\{([^}]+)\}", replace_var, template)
        return rendered

    def _format_evidence_list(self, elements: list) -> str:
        """Format evidence elements as markdown list.

        Args:
            elements: List of evidence elements

        Returns:
            Markdown-formatted list string
        """
        if not elements:
            return "- No elements found"

        formatted = []
        for elem in elements:
            if isinstance(elem, dict):
                # Try to extract meaningful representation
                elem_str = elem.get("description") or elem.get("type") or str(elem)
            else:
                elem_str = str(elem)
            formatted.append(f"- {elem_str}")

        return "\n".join(formatted)

    def _format_list(self, items: list[str], format_type: str) -> str:
        """Format a list of items according to output format.

        Args:
            items: List of string items
            format_type: Output format

        Returns:
            Formatted list string
        """
        if format_type == "html":
            items_html = "".join(f"<li>{item}</li>" for item in items)
            return f"<ul>{items_html}</ul>"
        else:  # markdown or plain_text
            return "\n".join(f"- {item}" for item in items)

    def _render_methodological_depth(
        self,
        config: dict[str, Any],
        evidence: dict[str, Any],
        validation: dict[str, Any],
        format_type: str,
    ) -> str:
        """Render methodological depth section with epistemological foundations.

        Transforms v3 contract’s methodological_depth into comprehensive documentation.

        Args:
            config: methodological_depth config from contract
            evidence: Evidence dict for contextualization
            validation: Validation dict
            format_type: Output format

        Returns:
            Formatted methodological depth documentation
        """
        if not config or "methods" not in config:
            return "Methodological documentation not available for this executor."

        sections = []

        # Header
        if format_type == "markdown":
            sections.append("#### Methodological Foundations\n")
        elif format_type == "html":
            sections.append("<h4>Methodological Foundations</h4>")
        else:
            sections.append("METHODOLOGICAL FOUNDATIONS\n")

        methods = config.get("methods", [])

        for method_info in methods:
            method_name = method_info.get("method_name", "Unknown")
            class_name = method_info.get("class_name", "Unknown")
            priority = method_info.get("priority", 0)
            role = method_info.get("role", "analysis")

            # Method header
            if format_type == "markdown":
                sections.append(
                    f"##### {class_name}.{method_name} (Priority {priority}, Role: {role})\n"
                )
            else:
                sections.append(
                    f"\n{class_name}.{method_name} (Priority {priority}, Role: {role})\n"
                )

            # Epistemological foundation
            epist = method_info.get("epistemological_foundation", {})
            if epist:
                sections.append(
                    self._render_epistemological_foundation(epist, format_type)
                )

            # Technical approach
            technical = method_info.get("technical_approach", {})
            if technical:
                sections.append(self._render_technical_approach(technical, format_type))

            # Output interpretation
            output_interp = method_info.get("output_interpretation", {})
            if output_interp:
                sections.append(
                    self._render_output_interpretation(output_interp, format_type)
                )

        # Method combination logic
        combination = config.get("method_combination_logic", {})
        if combination:
            sections.append(self._render_method_combination(combination, format_type))

        return "\n\n".join(filter(None, sections))

    def _render_epistemological_foundation(
        self, foundation: dict[str, Any], format_type: str
    ) -> str:
        """Render epistemological foundation section.

        Args:
            foundation: Epistemological foundation dict
            format_type: Output format

        Returns:
            Formatted epistemological foundation text
        """
        parts = []

        paradigm = foundation.get("paradigm")
        if paradigm:
            parts.append(f"**Paradigm**: {paradigm}")

        ontology = foundation.get("ontological_basis")
        if ontology:
            parts.append(f"**Ontological Basis**: {ontology}")

        stance = foundation.get("epistemological_stance")
        if stance:
            parts.append(f"**Epistemological Stance**: {stance}")

        framework = foundation.get("theoretical_framework", [])
        if framework:
            parts.append("**Theoretical Framework**:")
            for item in framework:
                parts.append(f"  - {item}")

        justification = foundation.get("justification")
        if justification:
            parts.append(f"**Justification**: {justification}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

    def _render_technical_approach(
        self, technical: dict[str, Any], format_type: str
    ) -> str:
        """Render technical approach section.

        Args:
            technical: Technical approach dict
            format_type: Output format

        Returns:
            Formatted technical approach text
        """
        parts = []

        method_type = technical.get("method_type")
        if method_type:
            parts.append(f"**Method Type**: {method_type}")

        algorithm = technical.get("algorithm")
        if algorithm:
            parts.append(f"**Algorithm**: {algorithm}")

        steps = technical.get("steps", [])
        if steps:
            parts.append("**Processing Steps**:")
            for step in steps:
                step_num = step.get("step", "?")
                step_name = step.get("name", "Unnamed")
                step_desc = step.get("description", "")
                parts.append(f"  {step_num}. **{step_name}**: {step_desc}")

        assumptions = technical.get("assumptions", [])
        if assumptions:
            parts.append("**Assumptions**:")
            for assumption in assumptions:
                parts.append(f"  - {assumption}")

        limitations = technical.get("limitations", [])
        if limitations:
            parts.append("**Limitations**:")
            for limitation in limitations:
                parts.append(f"  - {limitation}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

    def _render_output_interpretation(
        self, interpretation: dict[str, Any], format_type: str
    ) -> str:
        """Render output interpretation section.

        Args:
            interpretation: Output interpretation dict
            format_type: Output format

        Returns:
            Formatted output interpretation text
        """
        parts = []

        guide = interpretation.get("interpretation_guide", {})
        if guide:
            parts.append("**Interpretation Guide**:")
            for threshold_name, threshold_desc in guide.items():
                parts.append(f"  - **{threshold_name}**: {threshold_desc}")

        insights = interpretation.get("actionable_insights", [])
        if insights:
            parts.append("**Actionable Insights**:")
            for insight in insights:
                parts.append(f"  - {insight}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)

    def _render_method_combination(
        self, combination: dict[str, Any], format_type: str
    ) -> str:
        """Render method combination logic section.

        Args:
            combination: Method combination dict
            format_type: Output format
        Returns:
            Formatted method combination text
        """
        parts = []

        if format_type == "markdown":
            parts.append("#### Method Combination Strategy\n")
        else:
            parts.append("METHOD COMBINATION STRATEGY\n")

        strategy = combination.get("combination_strategy")
        if strategy:
            parts.append(f"**Strategy**: {strategy}")

        rationale = combination.get("rationale")
        if rationale:
            parts.append(f"**Rationale**: {rationale}")

        fusion = combination.get("evidence_fusion")
        if fusion:
            parts.append(f"**Evidence Fusion**: {fusion}")

        return "\n".join(parts) if format_type != "html" else "<br>".join(parts)


class DynamicContractExecutor(BaseExecutorWithContract):
    """Dynamic contract executor that accepts question_id at construction time.

    This executor enables the 300-contract model where each question has its own
    contract (Q001.v3.json through Q300.v3.json). Instead of requiring 300 subclasses,
    this single class can execute any contract by accepting the question_id parameter.

    The question_id is used to:
    1. Derive the base_slot (e.g., "Q001" -> "D1-Q1")
    2. Load the appropriate contract from executor_contracts/specialized/
    3. Execute the contract’s method_binding sequence

    Architecture Note:
    ==================
    OLD (30-executor multiplier pattern):
        - 30 executor classes (D1Q1_Executor through D6Q5_Executor)
        - Each executor answering 10 questions (multiplier pattern)
        - Required executors.py with hardcoded class definitions

    NEW (300-contract direct pattern):
        - Single DynamicContractExecutor class
        - 300 individual contracts (Q001.v3.json through Q300.v3.json)
        - Contract loaded dynamically by question_id

    Example:
        >>> executor = DynamicContractExecutor(
        ...     question_id="Q001",
        ...     method_executor=method_executor,
        ...     signal_registry=signal_registry,
        ...     config=config,
        ...     questionnaire_provider=questionnaire,
        ... )
        >>> result = executor.execute(document, method_executor, question_context=ctx)
    """

    # Class-level cache for question_id -> base_slot mapping
    _question_to_base_slot_cache: dict[str, str] = {}

    def __init__(
        self,
        method_executor: MethodExecutor,
        signal_registry: Any,
        config: Any,
        questionnaire_provider: Any,
        question_id: str,
        calibration_orchestrator: Any | None = None,
        enriched_packs: dict[str, Any] | None = None,
        validation_orchestrator: Any | None = None,
        calibration_policy: CalibrationPolicy | None = None,
    ) -> None:
        """Initialize dynamic contract executor for a specific question.

        Args:
            method_executor: MethodExecutor instance for method routing
            signal_registry: Signal registry for signal access
            config: ExecutorConfig for runtime parameters
            questionnaire_provider: Questionnaire provider
            question_id: Question identifier (e.g., "Q001", "Q150")
            calibration_orchestrator: Optional calibration orchestrator
            enriched_packs: Optional enriched signal packs
            validation_orchestrator: Optional validation orchestrator
            calibration_policy: Optional calibration policy
        """
        super().__init__(
            method_executor=method_executor,
            signal_registry=signal_registry,
            config=config,
            questionnaire_provider=questionnaire_provider,
            calibration_orchestrator=calibration_orchestrator,
            enriched_packs=enriched_packs,
            validation_orchestrator=validation_orchestrator,
            calibration_policy=calibration_policy,
        )
        self._question_id = question_id
        self._base_slot = self._derive_base_slot(question_id)

    @classmethod
    def _derive_base_slot(cls, question_id: str) -> str:
        """Derive base_slot from question_id.

        Conversion: Q001 -> D1-Q1, Q006 -> D2-Q1, Q031 -> D1-Q1 (for 6 dimensions Ã\227 5 questions per area)

        Args:
            question_id: Question identifier (e.g., "Q001", "Q150")

        Returns:
            Base slot string (e.g., "D1-Q1")
        """
        if question_id in cls._question_to_base_slot_cache:
            return cls._question_to_base_slot_cache[question_id]

        # Extract numeric part of question_id (e.g., "Q001" -> 1)
        try:
            q_number = int(question_id[1:])
        except (ValueError, IndexError):
            # Fallback: try to load contract and get base_slot from identity
            return cls._derive_base_slot_from_contract(question_id)

        # Calculate dimension and question within dimension
        # Assuming 6 dimensions Ã\227 5 questions per policy area Ã\227 10 policy areas = 300 questions
        # Pattern: D1-Q1 through D6-Q5, cycling through policy areas

        # Each "slot" covers 10 questions (one per policy area)
        slot_index = (q_number - 1) % 30  # 0-29 for the 30 slots
        dimension = (slot_index // 5) + 1  # 1-6
        question_in_dimension = (slot_index % 5) + 1  # 1-5

        base_slot = f"D{dimension}-Q{question_in_dimension}"
        cls._question_to_base_slot_cache[question_id] = base_slot

        return base_slot

    @classmethod
    def _derive_base_slot_from_contract(cls, question_id: str) -> str:
        """Fallback: derive base_slot by loading the contract’s identity.base_slot.

        Args:
            question_id: Question identifier

        Returns:
            Base slot from contract identity

        Raises:
            FileNotFoundError: If contract not found
        """
        contracts_dir = PROJECT_ROOT / "src" / "farfan_pipeline" / "phases" / "Phase_two" / "json_files_phase_two" / "executor_
contracts"

        # Try specialized contract
        v3_path = contracts_dir / "specialized" / f"{question_id}.v3.json"
        v2_path = contracts_dir / "specialized" / f"{question_id}.json"

        contract_path = v3_path if v3_path.exists() else v2_path
        if not contract_path.exists():
            raise FileNotFoundError(f"Contract not found for {question_id}")

        contract = json.loads(contract_path.read_text(encoding="utf-8"))
        base_slot = contract.get("identity", {}).get("base_slot", "D1-Q1")

        cls._question_to_base_slot_cache[question_id] = base_slot
        return base_slot

    @classmethod
    def get_base_slot(cls) -> str:
        """Get base slot - required by ABC but should use instance _base_slot.

        Note: This returns a default value for class-level operations.
        Instance-level operations should use self._base_slot.
        """
        # This is a slight hack - for dynamic executors, use instance._base_slot
        return "DYNAMIC"

    def _get_instance_base_slot(self) -> str:
        """Get the actual base_slot for this instance."""
        return self._base_slot

    def execute(
        self,
        document: PreprocessedDocument,
        method_executor: MethodExecutor,
        *,
        question_context: dict[str, Any],
    ) -> dict[str, Any]:
        """Execute the contract for this question.

        Overrides base to load contract using instance’s question_id.
        """
        if method_executor is not self.method_executor:
            raise RuntimeError(
                "Mismatched MethodExecutor instance for contract executor"
            )

        base_slot = self._base_slot
        if question_context.get("base_slot") and question_context.get("base_slot") != base_slot:
            # Allow mismatch if question_context uses the derived slot
            import logging
            logging.warning(
                f"Question base_slot {question_context.get('base_slot')} "
                f"differs from derived {base_slot}, using derived"
            )

        # Load contract using instance's question_id
        contract = self._load_contract(question_id=self._question_id)
        contract_version = contract.get("_contract_version", "v2")

        if contract_version == "v3":
            return self._execute_v3(document, question_context, contract)
        else:
            return self._execute_v2(document, question_context, contract)


# Export the dynamic executor
__all__ = ["BaseExecutorWithContract", "DynamicContractExecutor"]



###############################################################################
# FILE: phase2_c_carver.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_c_carver.py
###############################################################################

"""
Doctoral-Carver Narrative Synthesizer v3.0 (FULL EXTRACTION Edition)

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Carver Synthesizer
PHASE_ROLE: Generates PhD-level narrative responses with Raymond Carver minimalist style

==============================================================================

Genera respuestas PhD-level con estilo minimalista Raymond Carver:
- PrecisiÃ³n quirÃºrgica en cada afirmaciÃ³n
- Sin adornos retÃ³ricos vacÃos
- Cada palabra respaldada por evidencia
- Honestidad brutal sobre limitaciones
- Razonamiento causal explÃcito

CAMBIO MAYOR v3.0: ExtracciÃ³n COMPLETA del contrato v3
---------------------------------------------------------
v2.1 extraÃa solo: dimension, expected_elements, question_text, method_count, top 5 methods
v3.0 extrae TODO:
- 17 mÃ©todos con epistemological_foundation completo
- theoretical_framework con referencias acadÃ©micas
- technical_approach con algoritmos y limitaciones
- output_interpretation con actionable_insights
- assembly_flow de 5 pasos
- concrete_example como benchmark
- template_variable_bindings
- validation_against_expected_elements

Fundamentos TeÃ³ricos:
- Rhetorical Structure Theory (Mann & Thompson, 1988)
- Dempster-Shafer Evidence Theory (belief functions)
- Causal Inference Framework (Pearl, 2009)
- Argument Mining (Stab & Gurevych, 2017)
- Calibrated Uncertainty Quantification (Gneiting & Raftery, 2007)

Arquitectura v3.0:
1. EnhancedContractInterpreter: Extrae TODA la semÃ¡ntica del contrato v3
2. EvidenceAnalyzer: Construye grafo causal de evidencia
3. GapAnalyzer: AnÃ¡lisis multi-dimensional de vacÃos
4. BayesianConfidenceEngine: Inferencia calibrada de confianza
5. DimensionTheory: Estrategias teÃ³ricamente fundamentadas por D1-D6
6. DoctoralRenderer: Prosa con fundamentos epistemolÃ³gicos y citas
7. MacroSynthesizer: AgregaciÃ³n holÃstica con anÃ¡lisis PAÃ\227DIM

Invariantes:
[INV-001] Toda afirmaciÃ³n debe tener â\211¥1 evidencia citada
[INV-002] Gaps crÃticos siempre aparecen en respuesta
[INV-003] Confianza debe ser calibrada (no optimista)
[INV-004] Estilo Carver: oraciones cortas, verbos activos, sin adverbios
[INV-005] Macro synthesis con divergencia PAÃ\227DIM explÃcita
[INV-006] v3.0: Fundamentos epistemolÃ³gicos explÃcitos en sÃntesis doctoral
[INV-007] v3.0: Limitaciones metodolÃ³gicas como caveats honestos
[INV-008] v3.0: Referencias teÃ³ricas en formato acadÃ©mico

Author: F.A.R.F.A.N Pipeline
Version: 3.0.0-FULL-EXTRACTION
"""

from __future__ import annotations
```

Phase 2 Source Code

```python
import math
import re
import statistics
from abc import ABC, abstractmethod
from collections import defaultdict
from dataclasses import dataclass, field
from datetime import datetime, timezone
from enum import Enum
from typing import (
    Any,
    Dict,
    List,
    Optional,
    Tuple,
    TypeAlias,
)

# Readability and style checking libraries
try:
    import textstat  # Flesch-Kincaid and readability metrics
except ImportError:
    textstat = None  # type: ignore

try:
    from proselint.tools import lint as proselint_check  # Style and clarity checking
except ImportError:
    proselint_check = None  # type: ignore


# ============================================================================
# TYPE SYSTEM
# ============================================================================

Confidence: TypeAlias = float  # [0.0, 1.0]
BeliefMass: TypeAlias = float  # Dempster-Shafer belief
PlausibilityMass: TypeAlias = float  # Dempster-Shafer plausibility


class Dimension(Enum):
    """Las 6 dimensiones causales del modelo lÃ³gico."""
    D1_INSUMOS = "DIM01"      # Inputs: recursos, datos, diagnÃ³stico
    D2_ACTIVIDADES = "DIM02"  # Activities: acciones, instrumentos
    D3_PRODUCTOS = "DIM03"    # Outputs: entregables, metas
    D4_RESULTADOS = "DIM04"   # Outcomes: cambios inmediatos
    D5_IMPACTOS = "DIM05"     # Impacts: cambios largo plazo
    D6_CAUSALIDAD = "DIM06"   # Causality: lÃ³gica, M&E, adaptaciÃ³n


class EvidenceStrength(Enum):
    """Fuerza de evidencia segÃºn jerarquÃa epistemolÃ³gica."""
    DEFINITIVE = "definitive"      # Dato oficial verificable
    STRONG = "strong"              # MÃºltiples fuentes concordantes
    MODERATE = "moderate"          # Fuente Ãºnica confiable
    WEAK = "weak"                  # Inferido o parcial
    ABSENT = "absent"              # No encontrado


class GapSeverity(Enum):
    """Severidad de gaps con implicaciones para scoring."""
    CRITICAL = "critical"     # Bloquea evaluaciÃ³n positiva
    MAJOR = "major"           # Reduce score significativamente
    MINOR = "minor"           # Nota pero no bloquea
    COSMETIC = "cosmetic"     # Mejora deseable


class ArgumentRole(Enum):
    """Roles argumentativos (RST-inspired)."""
    CLAIM = "claim"           # AfirmaciÃ³n principal
    EVIDENCE = "evidence"     # Soporte factual
    WARRANT = "warrant"       # JustificaciÃ³n del vÃnculo
    QUALIFIER = "qualifier"   # LimitaciÃ³n/condiciÃ³n
    REBUTTAL = "rebuttal"     # Contraargumento reconocido
    BACKING = "backing"       # Soporte del warrant


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

@dataclass(frozen=True)
class ExpectedElement:
    """Elemento esperado con semÃ¡ntica completa."""
    type: str
    required: bool
    minimum: int
    category: str  # 'quantitative', 'qualitative', 'relational'
    weight: float  # Importancia relativa [0, 1]

    @classmethod
    def from_contract(cls, elem: Dict[str, Any]) -> ExpectedElement:
        """Factory desde contrato."""
        elem_type = elem.get("type", "")

        # Inferir categorÃa desde tipo
        quantitative_types = {
            "indicadores_cuantitativos", "series_temporales_aÃ±os",
            "monto_presupuestario", "meta_cuantificada", "linea_base",
            "porcentaje", "tasa", "indice"
        }
        relational_types = {
            "logica_causal_explicita", "ruta_transmision",
            "vinculo_causal", "dependencia_temporal"
        }

        if any(t in elem_type for t in quantitative_types):
            category = "quantitative"
        elif any(t in elem_type for t in relational_types):
            category = "relational"
        else:
            category = "qualitative"

        # Peso basado en required y minimum
        base_weight = 0.8 if elem.get("required", False) else 0.4
        min_val = elem.get("minimum", 0)
        weight = min(1.0, base_weight + (min_val * 0.05))

        return cls(
            type=elem_type,
            required=elem.get("required", False),
            minimum=elem.get("minimum", 1 if elem.get("required") else 0),
            category=category,
            weight=weight,
        )


@dataclass
class EvidenceItem:
    """Item de evidencia con metadatos ricos."""
    element_type: str
    value: Any
    confidence: float
    source_method: str
    document_location: Optional[str] = None

    # Computed properties
    strength: EvidenceStrength = EvidenceStrength.MODERATE
    is_quantitative: bool = False

    def __post_init__(self):
        """Compute derived properties."""
        # Determinar fuerza
        if self.confidence >= 0.9:
            self.strength = EvidenceStrength.STRONG
        elif self.confidence >= 0.7:
            self.strength = EvidenceStrength.MODERATE
        else:
            self.strength = EvidenceStrength.WEAK

        # Detectar si es cuantitativo
        if isinstance(self.value, (int, float)):
            self.is_quantitative = True
        elif isinstance(self.value, str):
            # Check for numeric patterns
            self.is_quantitative = bool(re.search(r'\d+[.,]?\d*\s*%?', self.value))


@dataclass
class EvidenceGap:
    """Gap con anÃ¡lisis causal de implicaciones."""
    element_type: str
    expected: int
    found: int
    severity: GapSeverity
    implication: str  # Por quÃ© importa este gap
    remediation: str  # QuÃ© harÃa falta

    @property
    def deficit(self) -> int:
        return max(0, self.expected - self.found)

    @property
    def fulfillment_ratio(self) -> float:
        if self.expected == 0:
            return 1.0
        return min(1.0, self.found / self.expected)


@dataclass
class ArgumentUnit:
    """Unidad argumentativa con rol retÃ³rico."""
    role: ArgumentRole
    content: str
    evidence_refs: List[str]  # IDs de evidencia que soportan
    confidence: float

    def render(self) -> str:
        """Render segÃºn rol."""
        if self.role == ArgumentRole.CLAIM:
            return self.content
        elif self.role == ArgumentRole.EVIDENCE:
            return f"- {self.content}"
        elif self.role == ArgumentRole.QUALIFIER:
            return f"*{self.content}*"
        elif self.role == ArgumentRole.REBUTTAL:
            return f"Sin embargo: {self.content}"
        return self.content


@dataclass
class BayesianConfidenceResult:
    """Resultado de inferencia bayesiana de confianza."""
    point_estimate: float
    belief: BeliefMass  # Grado de creencia
    plausibility: PlausibilityMass  # LÃmite superior de creencia
    uncertainty: float  # Ignorancia epistÃ©mica
    interval_95: Tuple[float, float]

    @property
    def is_calibrated(self) -> bool:
        """Check if interval is well-calibrated."""
        width = self.interval_95[1] - self.interval_95[0]
        return width >= 0.1  # No over-confident

    def to_label(self) -> str:
        """Human-readable label."""
        if self.point_estimate >= 0.85:
            return "ALTA"
        elif self.point_estimate >= 0.70:
            return "MEDIA-ALTA"
        elif self.point_estimate >= 0.50:
            return "MEDIA"
        elif self.point_estimate >= 0.30:
            return "BAJA"
        else:
            return "MUY BAJA"


@dataclass
class MethodEpistemology:
    """v3.0: Fundamentos epistemolÃ³gicos de un mÃ©todo."""
    method_name: str
    class_name: str
    priority: int
    role: str
    paradigm: str
    ontological_basis: str
    epistemological_stance: str
    theoretical_framework: List[str]
    justification: str
    method_type: str
    algorithm: str
    steps: List[Dict[str, Any]]
    assumptions: List[str]
    limitations: List[str]
    complexity: str
    output_structure: Dict[str, Any]
    interpretation_guide: Dict[str, str]
    actionable_insights: List[str]


@dataclass
class MethodologicalDepth:
    """v3.0: Profundidad metodolÃ³gica completa extraÃda del contrato."""
    methods: List[MethodEpistemology]
    total_methods: int
    paradigms_used: List[str]
    theoretical_references: List[str]
    all_limitations: List[str]
    all_assumptions: List[str]
    actionable_insights_by_method: Dict[str, List[str]]


@dataclass
class CarverAnswer:
    """Respuesta estructurada estilo Carver."""
    # Core components
    verdict: str  # Una oraciÃ³n. Directa. Sin escape.
    evidence_statements: List[str]  # Hechos. Verificables.
    gap_statements: List[str]  # VacÃos. Sin disculpas.

    # Confidence
    confidence_result: BayesianConfidenceResult
    confidence_statement: str

    # Metadata
    question_text: str
    dimension: Dimension
    method_note: str

    # Argumentative structure
    argument_units: List[ArgumentUnit] = field(default_factory=list)

    # v3.0: Enhanced doctoral components
    epistemology_section: str = ""
    limitations_section: str = ""
    benchmark_comparison: str = ""
    theoretical_references: List[str] = field(default_factory=list)
    # Trace
    synthesis_trace: Dict[str, Any] = field(default_factory=dict)

    # v3.0 Extensions (optional, backward compatible)
    methodological_depth: Optional["MethodologicalDepth"] = None
    limitations_statement: Optional[str] = None
    theoretical_references: Optional[List[str]] = None
    assumptions_statement: Optional[str] = None
    actionable_insights: Optional[List[str]] = None


@dataclass(frozen=True)
class MethodEpistemology:
    """Epistemological foundation of a method."""
    paradigm: str
    ontological_basis: str
    epistemological_stance: str
    theoretical_framework: List[str]
    justification: str


@dataclass(frozen=True)
class TechnicalApproach:
    """Technical approach and implementation details."""
    method_type: str
    algorithm: str
    steps: List[Dict[str, Any]]
    assumptions: List[str]
    limitations: List[str]
    complexity: str


@dataclass(frozen=True)
class OutputInterpretation:
    """Output structure and interpretation guidance."""
    output_structure: Dict[str, str]
    interpretation_guide: Dict[str, str]
    actionable_insights: List[str]


@dataclass(frozen=True)
class MethodDepthEntry:
    """Full methodological depth for a single method."""
    method_name: str
    class_name: str
    priority: int
    role: str
    epistemology: MethodEpistemology
    technical_approach: TechnicalApproach
    output_interpretation: OutputInterpretation


@dataclass(frozen=True)
class MethodCombinationLogic:
    """Logic for combining multiple methods."""
    dependency_graph: Dict[str, List[str]]
    trade_offs: List[str]
    evidence_fusion_approach: str


@dataclass(frozen=True)
class MethodologicalDepth:
    """Complete methodological depth from contract v3."""
    methods: List[MethodDepthEntry]
    combination_logic: Optional[MethodCombinationLogic]
    extraction_timestamp: str


# ============================================================================
# ENHANCED CONTRACT INTERPRETER (v3.0 - FULL EXTRACTION)
# ============================================================================

class EnhancedContractInterpreter:
    """
    v3.0: Extrae TODA la semÃ¡ntica del contrato v3 para sÃntesis doctoral.

    A diferencia del original que solo extraÃa metadata bÃ¡sica,
    este intÃ©rprete captura:
    - Fundamentos epistemolÃ³gicos de cada mÃ©todo
    - Frameworks teÃ³ricos y referencias acadÃ©micas
    - InterpretaciÃ³n de outputs con actionable insights
    - Limitaciones metodolÃ³gicas para caveats
    - Ejemplos concretos como benchmarks
    """

    # Mapeo de dimensiones a requisitos epistemolÃ³gicos
    DIMENSION_REQUIREMENTS = {
        Dimension.D1_INSUMOS: {
            "primary_need": "datos cuantitativos verificables",
            "evidence_type": "quantitative",
            "minimum_sources": 2,
            "temporal_requirement": True,
        },
        Dimension.D2_ACTIVIDADES: {
            "primary_need": "especificidad operativa",
            "evidence_type": "qualitative",
            "minimum_sources": 1,
            "temporal_requirement": False,
        },
        Dimension.D3_PRODUCTOS: {
            "primary_need": "proporcionalidad meta-problema",
            "evidence_type": "mixed",
            "minimum_sources": 1,
            "temporal_requirement": True,
        },
        Dimension.D4_RESULTADOS: {
            "primary_need": "indicadores medibles",
            "evidence_type": "quantitative",
            "minimum_sources": 1,
            "temporal_requirement": True,
        },
        Dimension.D5_IMPACTOS: {
            "primary_need": "teorÃa de cambio",
            "evidence_type": "relational",
            "minimum_sources": 1,
            "temporal_requirement": True,
        },
        Dimension.D6_CAUSALIDAD: {
            "primary_need": "lÃ³gica causal explÃcita",
            "evidence_type": "relational",
            "minimum_sources": 1,
            "temporal_requirement": False,
        },
    }

    @classmethod
    def extract_dimension(cls, contract: Dict) -> Dimension:
        """Extrae dimensiÃ³n con fallback inteligente."""
        identity = contract.get("identity", {})
        dim_id = identity.get("dimension_id", "")

        # Try direct match
        for dim in Dimension:
            if dim.value == dim_id:
                return dim

        # Fallback: infer from base_slot
        base_slot = identity.get("base_slot", "")
        if base_slot:
            try:
                dim_num = int(base_slot[1])  # "D1-Q1" -> 1
                return list(Dimension)[dim_num - 1]
            except (IndexError, ValueError):
                pass

        return Dimension.D1_INSUMOS  # Default

    @classmethod
    def extract_expected_elements(cls, contract: Dict) -> List[ExpectedElement]:
        """Extrae elementos con semÃ¡ntica enriquecida."""
        question_context = contract.get("question_context", {})
        raw_elements = question_context.get("expected_elements", [])

        return [ExpectedElement.from_contract(e) for e in raw_elements]

    @classmethod
    def extract_question_intent(cls, contract: Dict) -> Dict[str, Any]:
        """Extrae intenciÃ³n profunda de la pregunta."""
        question_context = contract.get("question_context", {})
        question_text = question_context.get("question_text", "")

        # Analizar tipo de pregunta
        q_lower = question_text.lower()

        if any(q in q_lower for q in ["Â¿cuÃ¡nto", "Â¿cuÃ¡ntos", "quÃ© porcentaje"]):
            question_type = "quantitative"
        elif any(q in q_lower for q in ["Â¿existe", "Â¿hay", "Â¿tiene", "Â¿incluye"]):
            question_type = "existence"
        elif any(q in q_lower for q in ["Â¿cÃ³mo", "Â¿de quÃ© manera"]):
            question_type = "process"
        elif any(q in q_lower for q in ["Â¿por quÃ©", "Â¿cuÃ¡l es la razÃ³n"]):
            question_type = "causal"
        else:
            question_type = "descriptive"

        # Extraer tema principal (policy area hint)
        policy_area = contract.get("identity", {}).get("policy_area_id", "")

        return {
            "question_text": question_text,
            "question_type": question_type,
            "policy_area": policy_area,
            "requires_numeric": question_type == "quantitative",
            "requires_causal_logic": question_type in ("causal", "process"),
        }

    @classmethod
    def get_dimension_theory(cls, dimension: Dimension) -> Dict[str, Any]:
        """Obtiene teorÃa epistemolÃ³gica de la dimensiÃ³n."""
        return cls.DIMENSION_REQUIREMENTS.get(dimension, {})

    @classmethod
```

Phase 2 Source Code

```
    def extract_method_metadata(cls, contract: Dict) -> Dict[str, Any]:
        """Extrae metadata bÃ¡sica de mÃ©todos (legacy compatibility)."""
        method_binding = contract.get("method_binding", {})

        return {
            "method_count": method_binding.get("method_count", 0),
            "orchestration_mode": method_binding.get("orchestration_mode", "unknown"),
            "methods": [
                m.get("method_name", "unknown")
                for m in method_binding.get("methods", [])
            ],
        }

    @classmethod
    def extract_methodological_depth(cls, contract: Dict) -> Optional[MethodologicalDepth]:
        """
        Extrae profundidad metodolÃ³gica completa del contrato v3.

        Extrae:
        - Fundamentos epistemolÃ³gicos de cada mÃ©todo
        - Enfoque tÃ©cnico y algoritmos
        - GuÃas de interpretaciÃ³n de salidas
        - LÃ³gica de combinaciÃ³n de mÃ©todos

        Returns:
            MethodologicalDepth si el contrato v3 tiene methodological_depth,
            None si es contrato v2 o falta el campo (backward compatible)
        """
        method_binding = contract.get("method_binding", {})
        methodological_depth_raw = method_binding.get("methodological_depth")

        if not methodological_depth_raw:
            return None

        methods_list = []
        methods_raw = methodological_depth_raw.get("methods", [])

        for method_raw in methods_raw:
            # Extract epistemology
            epi_raw = method_raw.get("epistemological_foundation", {})
            epistemology = MethodEpistemology(
                paradigm=epi_raw.get("paradigm", ""),
                ontological_basis=epi_raw.get("ontological_basis", ""),
                epistemological_stance=epi_raw.get("epistemological_stance", ""),
                theoretical_framework=epi_raw.get("theoretical_framework", []),
                justification=epi_raw.get("justification", "")
            )

            # Extract technical approach
            tech_raw = method_raw.get("technical_approach", {})
            technical_approach = TechnicalApproach(
                method_type=tech_raw.get("method_type", ""),
                algorithm=tech_raw.get("algorithm", ""),
                steps=tech_raw.get("steps", []),
                assumptions=tech_raw.get("assumptions", []),
                limitations=tech_raw.get("limitations", []),
                complexity=tech_raw.get("complexity", "")
            )

            # Extract output interpretation
            out_raw = method_raw.get("output_interpretation", {})
            output_interpretation = OutputInterpretation(
                output_structure=out_raw.get("output_structure", {}),
                interpretation_guide=out_raw.get("interpretation_guide", {}),
                actionable_insights=out_raw.get("actionable_insights", [])
            )

            # Create method depth entry
            method_entry = MethodDepthEntry(
                method_name=method_raw.get("method_name", ""),
                class_name=method_raw.get("class_name", ""),
                priority=method_raw.get("priority", 0),
                role=method_raw.get("role", ""),
                epistemology=epistemology,
                technical_approach=technical_approach,
                output_interpretation=output_interpretation
            )
            methods_list.append(method_entry)

        # Extract combination logic if present
        combination_logic = None
        combo_raw = methodological_depth_raw.get("method_combination_logic")
        if combo_raw:
            combination_logic = MethodCombinationLogic(
                dependency_graph=combo_raw.get("dependency_graph", {}),
                trade_offs=combo_raw.get("trade_offs", []),
                evidence_fusion_approach=combo_raw.get("evidence_fusion_approach", "")
            )

        # Create methodological depth
        return MethodologicalDepth(
            methods=methods_list,
            combination_logic=combination_logic,
            extraction_timestamp=datetime.now(timezone.utc).isoformat()
        )
```

Phase 2 Source Code

```
    @classmethod
    def extract_methodological_depth(cls, contract: Dict) -> MethodologicalDepth:
        """
        v3.0: Extrae la profundidad metodolÃ³gica COMPLETA de human_answer_structure.

        Returns:
            MethodologicalDepth con todos los mÃ©todos y sus fundamentos
        """
        human_answer = contract.get("human_answer_structure", {})
        method_binding = contract.get("method_binding", {})

        # Buscar methodological_depth en diferentes lugares
        methodological_depth = (
            method_binding.get("methodological_depth", {}) or
            human_answer.get("methodological_depth", {})
        )

        methods_data = methodological_depth.get("methods", [])

        extracted_methods: List[MethodEpistemology] = []
        all_paradigms: set[str] = set()
        all_references: List[str] = []
        all_limitations: List[str] = []
        all_assumptions: List[str] = []
        insights_by_method: Dict[str, List[str]] = {}

        for method in methods_data:
            epistemology = method.get("epistemological_foundation", {})
            technical = method.get("technical_approach", {})
            interpretation = method.get("output_interpretation", {})

            # Collect paradigm
            paradigm = epistemology.get("paradigm", "")
            if paradigm:
                all_paradigms.add(paradigm)

            # Collect theoretical references
            frameworks = epistemology.get("theoretical_framework", [])
            all_references.extend(frameworks)

            # Collect limitations
            limitations = technical.get("limitations", [])
            all_limitations.extend(limitations)

            # Collect assumptions
            assumptions = technical.get("assumptions", [])
            all_assumptions.extend(assumptions)

            # Collect actionable insights
            insights = interpretation.get("actionable_insights", [])
            method_name = method.get("method_name", "unknown")
            insights_by_method[method_name] = insights

            # Build MethodEpistemology
            extracted_methods.append(MethodEpistemology(
                method_name=method_name,
                class_name=method.get("class_name", ""),
                priority=method.get("priority", 0),
                role=method.get("role", ""),
                paradigm=paradigm,
                ontological_basis=epistemology.get("ontological_basis", ""),
                epistemological_stance=epistemology.get("epistemological_stance", ""),
                theoretical_framework=frameworks,
                justification=epistemology.get("justification", ""),
                method_type=technical.get("method_type", ""),
                algorithm=technical.get("algorithm", ""),
                steps=technical.get("steps", []),
                assumptions=assumptions,
                limitations=limitations,
                complexity=technical.get("complexity", ""),
                output_structure=interpretation.get("output_structure", {}),
                interpretation_guide=interpretation.get("interpretation_guide", {}),
                actionable_insights=insights,
            ))

        return MethodologicalDepth(
            methods=extracted_methods,
            total_methods=len(extracted_methods),
            paradigms_used=list(all_paradigms),
            theoretical_references=list(set(all_references)),
            all_limitations=list(set(all_limitations)),
            all_assumptions=list(set(all_assumptions)),
            actionable_insights_by_method=insights_by_method,
        )

    @classmethod
    def extract_assembly_flow(cls, contract: Dict) -> Dict[str, str]:
        """v3.0: Extrae el flujo de ensamblaje de 5 pasos."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("assembly_flow", {})

    @classmethod
    def extract_evidence_schema(cls, contract: Dict) -> Dict[str, Any]:
        """v3.0: Extrae el schema completo de estructura de evidencia."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("evidence_structure_schema", {})
    @classmethod
    def extract_concrete_example(cls, contract: Dict) -> Dict[str, Any]:
        """v3.0: Extrae el ejemplo concreto como benchmark."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("concrete_example", {})

    @classmethod
    def extract_template_variables(cls, contract: Dict) -> Dict[str, str]:
        """v3.0: Extrae las variables disponibles para templates."""
        human_answer = contract.get("human_answer_structure", {})
        bindings = human_answer.get("template_variable_bindings", {})
        return bindings.get("variables", {})

    @classmethod
    def extract_validation_mapping(cls, contract: Dict) -> Dict[str, Any]:
        """v3.0: Extrae el mapping de validaciÃ³n esperado â\206\222 encontrado."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("validation_against_expected_elements", {})

    @classmethod
    def extract_usage_notes(cls, contract: Dict) -> Dict[str, str]:
        """v3.0: Extrae notas de uso para diferentes roles."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("usage_notes", {})

    @classmethod
    def extract_evidence_graph_structure(cls, contract: Dict) -> Dict[str, Any]:
        """v3.0: Extrae la estructura del grafo de evidencia post-Nexus."""
        human_answer = contract.get("human_answer_structure", {})
        return human_answer.get("evidence_structure_post_nexus", {})


# ============================================================================
# EVIDENCE ANALYZER
# ============================================================================

class EvidenceAnalyzer:
    """
    AnÃ¡lisis profundo de evidencia con construcciÃ³n de grafo causal.
    """

    @staticmethod
    def extract_items(evidence: Dict[str, Any]) -> List[EvidenceItem]:
        """Extrae items de evidencia estructurados."""
        items = []

        for elem in evidence.get("elements", []):
            if isinstance(elem, dict):
                items.append(EvidenceItem(
                    element_type=elem.get("type", "unknown"),
                    value=elem.get("value", elem.get("description", "")),
                    confidence=float(elem.get("confidence", 0.5)),
                    source_method=elem.get("source_method", "unknown"),
                    document_location=elem.get("page", elem.get("location")),
                ))

        return items

    @staticmethod
    def count_by_type(items: List[EvidenceItem]) -> Dict[str, int]:
        """Cuenta items por tipo."""
        counts: Dict[str, int] = defaultdict(int)
        for item in items:
            counts[item.element_type] += 1
        return dict(counts)

    @staticmethod
    def group_by_type(items: List[EvidenceItem]) -> Dict[str, List[EvidenceItem]]:
        """Agrupa items por tipo."""
        groups: Dict[str, List[EvidenceItem]] = defaultdict(list)
        for item in items:
            groups[item.element_type].append(item)
        return dict(groups)

    @staticmethod
    def analyze_strength_distribution(items: List[EvidenceItem]) -> Dict[str, int]:
        """Analiza distribuciÃ³n de fuerza de evidencia."""
        distribution: Dict[str, int] = defaultdict(int)
        for item in items:
            distribution[item.strength.value] += 1
        return dict(distribution)

    @staticmethod
    def find_corroborations(items: List[EvidenceItem]) -> List[Tuple[EvidenceItem, EvidenceItem]]:
        """
        Encuentra pares de evidencia que se corroboran.

        CorroboraciÃ³n: mismo tipo, diferentes fuentes, valores consistentes.
        """
        corroborations = []
        groups = EvidenceAnalyzer.group_by_type(items)

        for elem_type, group_items in groups.items():
            if len(group_items) < 2:
                continue
            # Check pairs
            for i, item1 in enumerate(group_items):
                for item2 in group_items[i+1:]:
                    if item1.source_method != item2.source_method:
                        # Different sources = potential corroboration
                        corroborations.append((item1, item2))

        return corroborations

    @staticmethod
    def find_contradictions(items: List[EvidenceItem]) -> List[Tuple[EvidenceItem, EvidenceItem, str]]:
        """
        Encuentra contradicciones en evidencia.

        Returns: List of (item1, item2, explanation)
        """
        contradictions = []
        groups = EvidenceAnalyzer.group_by_type(items)

        for elem_type, group_items in groups.items():
            if len(group_items) < 2:
                continue

            # Check for numeric contradictions
            numeric_items = [i for i in group_items if i.is_quantitative]
            if len(numeric_items) >= 2:
                values = []
                for item in numeric_items:
                    try:
                        # Extract numeric value
                        val_str = str(item.value)
                        nums = re.findall(r'[\d.]+', val_str)
                        if nums:
                            values.append((item, float(nums[0])))
                    except (ValueError, TypeError, IndexError):
                        pass

                if len(values) >= 2:
                    # Check for significant divergence (>50% difference)
                    for i, (item1, val1) in enumerate(values):
                        for item2, val2 in values[i+1:]:
                            if val1 > 0 and abs(val1 - val2) / val1 > 0.5:
                                contradictions.append((
                                    item1, item2,
                                    f"Divergencia numÃ©rica: {val1} vs {val2}"
                                ))

        return contradictions


# ============================================================================
# GAP ANALYZER
# ============================================================================

class GapAnalyzer:
    """
    AnÃ¡lisis multi-dimensional de gaps con implicaciones causales.
    """

    # Implicaciones por tipo de elemento faltante
    GAP_IMPLICATIONS = {
        "fuentes_oficiales": (
            "Sin fuentes oficiales, la credibilidad del diagnÃ³stico es cuestionable.",
            "Citar fuentes como DANE, Medicina Legal, ICBF."
        ),
        "indicadores_cuantitativos": (
            "Sin indicadores numÃ©ricos, no hay lÃnea base medible.",
            "Incluir tasas, porcentajes o valores absolutos con fuente."
        ),
        "series_temporales_aÃ±os": (
            "Sin series temporales, no se puede evaluar tendencia.",
            "Presentar datos de al menos 3 aÃ±os consecutivos."
        ),
        "cobertura_territorial_especificada": (
            "Sin especificaciÃ³n territorial, el alcance es ambiguo.",
            "Definir si es municipal, departamental o por zonas."
        ),
        "logica_causal_explicita": (
            "Sin lÃ³gica causal, la teorÃa de cambio es invisible.",
            "Explicitar cadena: insumo â\206\222 actividad â\206\222 producto â\206\222 resultado."
        ),
        "poblacion_objetivo_definida": (
            "Sin poblaciÃ³n objetivo, no hay focalizaciÃ³n.",
            "Definir grupo beneficiario con caracterÃsticas especÃficas."
        ),
        "instrumento_especificado": (
            "Sin instrumentos, las actividades son abstractas.",
            "Nombrar programas, proyectos o mecanismos concretos."
        ),
        "meta_cuantificada": (
            "Sin metas cuantificadas, no hay accountability.",
            "Establecer valores objetivo con plazo."
        ),
        "linea_base_resultado": (
            "Sin lÃnea base, no se puede medir avance.",
            "Documentar situaciÃ³n inicial con fecha."
        ),
        "impacto_definido": (
            "Sin impactos definidos, el propÃ³sito final es difuso.",
            "Describir cambios de largo plazo esperados."
        ),
        "sistema_monitoreo": (
            "Sin sistema de monitoreo, no hay seguimiento.",
            "Especificar indicadores, frecuencia y responsables."
        ),
    }

    @classmethod
    def identify_gaps(
        cls,
        expected: List[ExpectedElement],
        found_counts: Dict[str, int],
        dimension: Dimension,
    ) -> List[EvidenceGap]:
        """
        Identifica gaps con severidad calibrada por dimensiÃ³n.
        """
        gaps = []
        dim_theory = EnhancedContractInterpreter.get_dimension_theory(dimension)

        for elem in expected:
            found = found_counts.get(elem.type, 0)

            if found >= elem.minimum:
                continue  # No gap

            # Determinar severidad
            severity = cls._compute_severity(elem, found, dim_theory)

            # Obtener implicaciÃ³n y remediaciÃ³n
            implication, remediation = cls.GAP_IMPLICATIONS.get(
                elem.type,
                (f"Falta {elem.type}.", f"Agregar {elem.type}.")
            )

            gaps.append(EvidenceGap(
                element_type=elem.type,
                expected=elem.minimum,
                found=found,
                severity=severity,
                implication=implication,
                remediation=remediation,
            ))

        # Sort by severity
        severity_order = {
            GapSeverity.CRITICAL: 0,
            GapSeverity.MAJOR: 1,
            GapSeverity.MINOR: 2,
            GapSeverity.COSMETIC: 3,
        }
        gaps.sort(key=lambda g: severity_order[g.severity])

        return gaps

    @classmethod
    def _compute_severity(
        cls,
        elem: ExpectedElement,
        found: int,
        dim_theory: Dict[str, Any],
    ) -> GapSeverity:
        """Computa severidad basada en contexto dimensional."""

        # Critical if required and completely missing
        if elem.required and found == 0:
            return GapSeverity.CRITICAL

        # Check if matches dimension’s primary need
        evidence_type = dim_theory.get("evidence_type", "")

        # Critical if element type matches dimension’s evidence type and missing
        if elem.category == evidence_type and found == 0:
            return GapSeverity.CRITICAL

        # Major if required but partial
        if elem.required and found < elem.minimum:
            return GapSeverity.MAJOR

        # Major if high weight and missing
        if elem.weight >= 0.7 and found == 0:
            return GapSeverity.MAJOR

        # Minor for optional but expected
        if elem.minimum > 0 and found < elem.minimum:
            return GapSeverity.MINOR

        return GapSeverity.COSMETIC


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

Phase 2 Source Code

```
# BAYESIAN CONFIDENCE ENGINE
# ==============================================================================

class BayesianConfidenceEngine:
    """
    Inferencia bayesiana de confianza con calibraciÃ³n.

    Usa Dempster-Shafer para manejar incertidumbre epistÃ©mica.
    """

    @staticmethod
    def compute(
        items: List[EvidenceItem],
        gaps: List[EvidenceGap],
        corroborations: List[Tuple[EvidenceItem, EvidenceItem]],
        contradictions: List[Tuple[EvidenceItem, EvidenceItem, str]],
    ) -> BayesianConfidenceResult:
        """
        Computa confianza calibrada usando Dempster-Shafer.
        """
        if not items:
            return BayesianConfidenceResult(
                point_estimate=0.0,
                belief=0.0,
                plausibility=0.3,
                uncertainty=1.0,
                interval_95=(0.0, 0.3),
            )

        # 1. Base: average confidence of evidence
        confidences = [i.confidence for i in items]
        base_conf = statistics.mean(confidences)

        # 2. Boost for corroborations
        corroboration_boost = min(0.15, len(corroborations) * 0.05)

        # 3. Penalty for contradictions
        contradiction_penalty = min(0.25, len(contradictions) * 0.1)

        # 4. Penalty for gaps
        critical_gaps = sum(1 for g in gaps if g.severity == GapSeverity.CRITICAL)
        major_gaps = sum(1 for g in gaps if g.severity == GapSeverity.MAJOR)
        gap_penalty = min(0.4, critical_gaps * 0.15 + major_gaps * 0.05)

        # 5. Compute belief mass (lower bound of confidence)
        belief = max(0.0, base_conf + corroboration_boost - contradiction_penalty - gap_penalty)
        belief = belief * (1 - 0.1 * critical_gaps)  # Further reduce for critical gaps

        # 6. Compute plausibility (upper bound)
        plausibility = min(1.0, belief + 0.2)

        # 7. Epistemic uncertainty
        uncertainty = plausibility - belief

        # 8. Point estimate (expected value under ignorance)
        pessimism_weight = 0.6  # Be conservative
        point_estimate = pessimism_weight * belief + (1 - pessimism_weight) * plausibility

        # 9. Calibrated interval using Wilson score
        n = len(items)
        z = 1.96  # 95% CI

        p = point_estimate
        denominator = 1 + z**2 / n
        center = (p + z**2 / (2*n)) / denominator
        margin = z * math.sqrt((p * (1 - p) + z**2 / (4*n)) / n) / denominator

        lower = max(0.0, center - margin - gap_penalty)
        upper = min(1.0, center + margin)

        return BayesianConfidenceResult(
            point_estimate=round(point_estimate, 3),
            belief=round(belief, 3),
            plausibility=round(plausibility, 3),
            uncertainty=round(uncertainty, 3),
            interval_95=(round(lower, 3), round(upper, 3)),
        )


# ==============================================================================
# DIMENSION-SPECIFIC STRATEGIES
# ==============================================================================

class DimensionStrategy(ABC):
    """Base class for dimension-specific strategies."""

    @property
    @abstractmethod
    def dimension(self) -> Dimension:
        pass

    @abstractmethod
    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        """Prefix for verdict based on dimension theory."""
        pass
```

Phase 2 Source Code

```
    @abstractmethod
    def key_requirement(self) -> str:
        """Key requirement for this dimension."""
        pass

    @abstractmethod
    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        """Dimension-specific confidence interpretation."""
        pass


class D1InsumosStrategy(DimensionStrategy):
    """D1: Insumos - DiagnÃ³stico y datos cuantitativos."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D1_INSUMOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "El diagnÃ³stico carece de fundamento cuantitativo."
        return "El diagnÃ³stico tiene base cuantitativa."

    def key_requirement(self) -> str:
        return "Datos numÃ©ricos de fuentes oficiales."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "Los datos son verificables."
        return "Faltan datos verificables."


class D2ActividadesStrategy(DimensionStrategy):
    """D2: Actividades - Especificidad operativa."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D2_ACTIVIDADES

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "Las actividades son vagas."
        return "Las actividades estÃ¡n especificadas."

    def key_requirement(self) -> str:
        return "Instrumento, poblaciÃ³n y lÃ³gica definidos."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La especificaciÃ³n es operativa."
        return "Falta especificidad operativa."


class D3ProductosStrategy(DimensionStrategy):
    """D3: Productos - Proporcionalidad y metas."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D3_PRODUCTOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "Los productos no son proporcionales al problema."
        return "Los productos son proporcionales."

    def key_requirement(self) -> str:
        return "Metas cuantificadas y proporcionales."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La proporcionalidad es clara."
        return "La proporcionalidad es cuestionable."


class D4ResultadosStrategy(DimensionStrategy):
    """D4: Resultados - Indicadores de outcome."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D4_RESULTADOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "Los resultados no son medibles."
        return "Los resultados tienen indicadores."

    def key_requirement(self) -> str:
        return "Indicadores con lÃ-nea base y meta."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "Los indicadores permiten seguimiento."
        return "El seguimiento no es posible."


class D5ImpactosStrategy(DimensionStrategy):
    """D5: Impactos - Cambios de largo plazo."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D5_IMPACTOS

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "El impacto de largo plazo no estÃ¡ definido."
        return "El impacto estÃ¡ conceptualizado."

    def key_requirement(self) -> str:
        return "TeorÃa de cambio con horizonte temporal."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "La teorÃa de cambio es plausible."
        return "La teorÃa de cambio es dÃ©bil."


class D6CausalidadStrategy(DimensionStrategy):
    """D6: Causalidad - M&E y adaptaciÃ³n."""

    @property
    def dimension(self) -> Dimension:
        return Dimension.D6_CAUSALIDAD

    def verdict_prefix(self, has_critical_gaps: bool) -> str:
        if has_critical_gaps:
            return "La lÃ³gica causal no es explÃcita."
        return "La cadena causal estÃ¡ documentada."

    def key_requirement(self) -> str:
        return "Sistema de M&E con ciclos de aprendizaje."

    def interpret_confidence(self, conf: BayesianConfidenceResult) -> str:
        if conf.point_estimate >= 0.7:
            return "El sistema permite adaptaciÃ³n."
        return "No hay mecanismo de correcciÃ³n."


def get_dimension_strategy(dimension: Dimension) -> DimensionStrategy:
    """Factory for dimension strategies."""
    strategies = {
        Dimension.D1_INSUMOS: D1InsumosStrategy(),
        Dimension.D2_ACTIVIDADES: D2ActividadesStrategy(),
        Dimension.D3_PRODUCTOS: D3ProductosStrategy(),
        Dimension.D4_RESULTADOS: D4ResultadosStrategy(),
        Dimension.D5_IMPACTOS: D5ImpactosStrategy(),
        Dimension.D6_CAUSALIDAD: D6CausalidadStrategy(),
    }
    return strategies.get(dimension, D1InsumosStrategy())


# ============================================================================
# READABILITY & STYLE CHECKER (Flesch-Kincaid + Proselint)
# ============================================================================

@dataclass
class ReadabilityMetrics:
    """MÃ©tricas de legibilidad segÃºn Flesch-Kincaid y Proselint."""
    flesch_reading_ease: Optional[float] = None
    flesch_kincaid_grade: Optional[float] = None
    gunning_fog: Optional[float] = None
    avg_sentence_length: Optional[float] = None
    avg_word_length: Optional[float] = None
    proselint_errors: List[Dict[str, Any]] = field(default_factory=list)
    proselint_score: Optional[float] = None

    def passes_carver_standards(self) -> bool:
        """Verifica si el texto cumple estÃ¡ndares Carver."""
        if self.flesch_reading_ease and self.flesch_reading_ease < 60:
            return False
        if self.flesch_kincaid_grade and self.flesch_kincaid_grade > 12:
            return False
        if self.avg_sentence_length and self.avg_sentence_length > 20:
            return False
        if self.proselint_score and self.proselint_score < 0.9:
            return False
        return True


class ReadabilityChecker:
    """Aplica Flesch-Kincaid y Proselint para garantizar claridad Carver."""

    @staticmethod
    def check_text(text: str) -> ReadabilityMetrics:
        """Analiza texto con Flesch-Kincaid y Proselint."""
        metrics = ReadabilityMetrics()

        if textstat:
            try:
                metrics.flesch_reading_ease = textstat.flesch_reading_ease(text)
                metrics.flesch_kincaid_grade = textstat.flesch_kincaid_grade(text)
                metrics.gunning_fog = textstat.gunning_fog(text)
                sentences = textstat.sentence_count(text)
                words = textstat.lexicon_count(text, removepunct=True)
                if sentences > 0:
                    metrics.avg_sentence_length = words / sentences

                chars = sum(len(word) for word in text.split())
                if words > 0:
                    metrics.avg_word_length = chars / words

            except Exception:
                pass

        if proselint_check:
            try:
                errors = proselint_check(text)
                if errors:
                    metrics.proselint_errors = errors
                    metrics.proselint_score = max(0.0, 1.0 - len(errors) / 100.0)
                else:
                    metrics.proselint_score = 1.0
            except Exception:
                metrics.proselint_score = 1.0

        return metrics

    @staticmethod
    def enforce_carver_style(text: str) -> Tuple[str, ReadabilityMetrics]:
        """Analiza y opcionalmente ajusta texto para cumplir estÃ¡ndares Carver."""
        metrics = ReadabilityChecker.check_text(text)

        if metrics.passes_carver_standards():
            return text, metrics

        adjusted_text = text

        if metrics.avg_sentence_length and metrics.avg_sentence_length > 20:
            adjusted_text = re.sub(r',\s+([a-zÃ¡Ã©ÃÃ³ÃºÃ±])', r'. \1', adjusted_text)
            adjusted_text = re.sub(r'\s+y\s+([a-zÃ¡Ã©ÃÃ³ÃºÃ±])', r'. \1', adjusted_text)

        metrics = ReadabilityChecker.check_text(adjusted_text)

        return adjusted_text, metrics


# ============================================================================
# DOCTORAL RENDERER (v3.0 - Enhanced with Epistemology)
# ============================================================================

class DoctoralRenderer:
    """
    v3.0: Renderiza prosa estilo Raymond Carver con fundamentos doctorales.

    Principios:
    - Oraciones cortas. Sujeto-verbo-objeto.
    - Verbos activos. Sin pasiva.
    - Sin adverbios. Sin adjetivos innecesarios.
    - Cada palabra cuenta. Si sobra, eliminar.
    - La verdad es suficiente. Sin adornos.

    Nuevos principios v3.0:
    - Fundamentos epistemolÃ³gicos explÃcitos cuando relevante
    - Referencias teÃ³ricas en formato acadÃ©mico
    - Limitaciones como caveats honestos
    - Benchmark comparison cuando disponible
    """

    TYPE_LABELS = {
        "fuentes_oficiales": "fuentes oficiales",
        "indicadores_cuantitativos": "indicadores numÃ©ricos",
        "series_temporales_aÃ±os": "series temporales",
        "cobertura_territorial_especificada": "cobertura territorial",
        "instrumento_especificado": "instrumentos",
        "poblacion_objetivo_definida": "poblaciÃ³n objetivo",
        "logica_causal_explicita": "lÃ³gica causal",
        "riesgos_identificados": "riesgos",
        "mitigacion_propuesta": "mitigaciÃ³n",
        "impacto_definido": "impactos",
        "rezago_temporal": "horizonte temporal",
        "ruta_transmision": "ruta de transmisiÃ³n",
        "proporcionalidad_meta_problema": "proporcionalidad",
        "linea_base_resultado": "lÃnea base",
        "meta_resultado": "metas",
        "meta_cuantificada": "metas cuantificadas",
        "metrica_outcome": "mÃ©tricas",
        "sistema_monitoreo": "sistema de monitoreo",
        "ciclos_aprendizaje": "ciclos de aprendizaje",
        "mecanismos_correccion": "mecanismos de correcciÃ³n",
        "analisis_contextual": "anÃ¡lisis contextual",
        "enfoque_diferencial": "enfoque diferencial",
    }

    @classmethod
    def humanize(cls, elem_type: str) -> str:
        """Convert technical type to plain Spanish."""
        return cls.TYPE_LABELS.get(elem_type, elem_type.replace("_", " "))
```

Phase 2 Source Code

```
    @classmethod
    def render_verdict(
        cls,
        strategy: DimensionStrategy,
        gaps: List[EvidenceGap],
        items: List[EvidenceItem],
    ) -> str:
        """Render verdict: una oraciÃ³n. Sin escape."""
        critical_gaps = [g for g in gaps if g.severity == GapSeverity.CRITICAL]
        has_critical = len(critical_gaps) > 0

        prefix = strategy.verdict_prefix(has_critical)

        if not items:
            return f"{prefix} No hay evidencia."

        if has_critical:
            missing = [cls.humanize(g.element_type) for g in critical_gaps[:2]]
            return f"{prefix} Falta: {’, ’.join(missing)}."

        return prefix

    @classmethod
    def render_evidence_statements(
        cls,
        items: List[EvidenceItem],
        found_counts: Dict[str, int],
    ) -> List[str]:
        """Render evidence as facts. Short. Verifiable."""
        statements = []

        total = len(items)
        if total > 0:
            statements.append(f"{total} elementos de evidencia.")

        sorted_types = sorted(found_counts.items(), key=lambda x: x[1], reverse=True)[:3]
        for elem_type, count in sorted_types:
            label = cls.humanize(elem_type)
            statements.append(f"{count} {label}.")

        strong = sum(1 for i in items if i.strength == EvidenceStrength.STRONG)
        if strong > 0:
            statements.append(f"{strong} elementos con alta confianza.")

        return statements

    @classmethod
    def render_gap_statements(cls, gaps: List[EvidenceGap]) -> List[str]:
        """Render gaps. No excuses. Just facts."""
        statements = []

        for gap in gaps[:4]:
            label = cls.humanize(gap.element_type)

            if gap.found == 0:
                statements.append(f"No hay {label}.")
            else:
                statements.append(f"{gap.found} {label}. Se necesitan {gap.expected}.")

        return statements

    @classmethod
    def render_confidence_statement(
        cls,
        conf: BayesianConfidenceResult,
        strategy: DimensionStrategy,
    ) -> str:
        """Render confidence. Honest. Calibrated."""
        label = conf.to_label()
        pct = int(conf.point_estimate * 100)

        interpretation = strategy.interpret_confidence(conf)

        return f"Confianza {label} ({pct}%). {interpretation}"

    @classmethod
    def render_method_note(cls, method_meta: Dict[str, Any]) -> str:
        """Render method note. Brief. At the end."""
        count = method_meta.get("method_count", 0)
        return f"AnÃ¡lisis con {count} mÃ©todos."

    @classmethod
    def render_limitations_section(cls, methodological_depth: MethodologicalDepth) -> str:
        """
        Render limitations section.Max 5.Carver style.

        Extrae limitaciones de technical_approach de cada mÃ©todo,
        deduplica y prioriza las mÃ¡s relevantes.
        """
        all_limitations = []
        for method in methodological_depth.methods:
            for limitation in method.technical_approach.limitations:
                if limitation and limitation not in all_limitations:
                    all_limitations.append(limitation)

        if not all_limitations:
            return ""

        # Max 5 limitations
        selected = all_limitations[:5]

        lines = ["## Limitaciones\n"]
        for lim in selected:
            lines.append(f"- {lim}")

        return "\n".join(lines)

    @classmethod
    def render_theoretical_references(cls, methodological_depth: MethodologicalDepth) -> str:
        """
        Render theoretical references.Deduplicated.Max 6.

        Extrae referencias del theoretical_framework de cada mÃ©todo.
        """
        all_refs = []
        for method in methodological_depth.methods:
            for ref in method.epistemology.theoretical_framework:
                if ref and ref not in all_refs:
                    all_refs.append(ref)

        if not all_refs:
            return ""

        # Max 6 references
        selected = all_refs[:6]

        lines = ["## Referencias TeÃ³ricas\n"]
        for i, ref in enumerate(selected, 1):
            lines.append(f"{i}. {ref}")

        return "\n".join(lines)

    @classmethod
    def render_actionable_insights(cls, methodological_depth: MethodologicalDepth) -> str:
        """
        Render actionable insights.Prioritized by relevance.

        Extrae insights de output_interpretation de cada mÃ©todo.
        """
        all_insights = []
        for method in methodological_depth.methods:
            for insight in method.output_interpretation.actionable_insights:
                if insight and insight not in all_insights:
                    all_insights.append(insight)

        if not all_insights:
            return ""

        lines = ["## Insights Accionables\n"]
        for insight in all_insights[:6]:  # Max 6
            lines.append(f"- {insight}")

        return "\n".join(lines)

    @classmethod
    def render_assumptions_section(cls, methodological_depth: MethodologicalDepth) -> str:
        """
        Render assumptions section.Deduplicated.Max 5.

        Extrae assumptions de technical_approach de cada mÃ©todo.
        """
        all_assumptions = []
        for method in methodological_depth.methods:
            for assumption in method.technical_approach.assumptions:
                if assumption and assumption not in all_assumptions:
                    all_assumptions.append(assumption)

        if not all_assumptions:
            return ""

        # Max 5 assumptions
        selected = all_assumptions[:5]

        lines = ["## Supuestos MetodolÃ³gicos\n"]
        for assum in selected:
            lines.append(f"- {assum}")

        return "\n".join(lines)

    @classmethod
    def render_full_answer(cls, answer: CarverAnswer) -> str:
        """
        Render complete answer in Carver style with readability enforcement.
        """
        sections = []

        # Question context
        sections.append(f"**Pregunta**: {answer.question_text}\n")

        # Verdict (the core)
        sections.append(f"## Respuesta\n\n{answer.verdict}\n")
```

Phase 2 Source Code

```
        # Evidence (facts only)
        if answer.evidence_statements:
            sections.append("## Evidencia\n")
            for stmt in answer.evidence_statements:
                sections.append(f"- {stmt}")

        # Gaps (if any)
        if answer.gap_statements:
            sections.append("\n## VacÃos\n")
            for stmt in answer.gap_statements:
                sections.append(f"- {stmt}")

        # Confidence
        sections.append(f"\n## Confianza\n\n{answer.confidence_statement}")

        # v3.0 Extensions (only if present)
        if answer.methodological_depth:
            # Limitations
            limitations_text = cls.render_limitations_section(answer.methodological_depth)
            if limitations_text:
                sections.append(f"\n{limitations_text}")

            # Assumptions
            assumptions_text = cls.render_assumptions_section(answer.methodological_depth)
            if assumptions_text:
                sections.append(f"\n{assumptions_text}")

            # Actionable Insights
            insights_text = cls.render_actionable_insights(answer.methodological_depth)
            if insights_text:
                sections.append(f"\n{insights_text}")

            # Theoretical References
            refs_text = cls.render_theoretical_references(answer.methodological_depth)
            if refs_text:
                sections.append(f"\n{refs_text}")

        # Method note (discrete)
        sections.append(f"\n---\n*{answer.method_note}*")

        # Join all sections
        full_text = "\n".join(sections)

        # Apply Flesch-Kincaid and Proselint readability checking
        adjusted_text, metrics = ReadabilityChecker.enforce_carver_style(full_text)

        # Add readability report if metrics available
        if metrics.flesch_reading_ease or metrics.proselint_score:
            readability_note = "\n\n---\n**MÃ©tricas de Legibilidad**:\n"
            if metrics.flesch_reading_ease:
                readability_note += f"- Flesch Reading Ease: {metrics.flesch_reading_ease:.1f} "
                readability_note += ("(FÃ¡cil)" if metrics.flesch_reading_ease >= 60 else "(DifÃcil)")
                readability_note += "\n"
            if metrics.flesch_kincaid_grade:
                readability_note += f"- Nivel Educativo: {metrics.flesch_kincaid_grade:.1f} grado\n"
            if metrics.avg_sentence_length:
                readability_note += f"- Longitud Promedio: {metrics.avg_sentence_length:.1f} palabras/oraciÃ³n\n"
            if metrics.proselint_score is not None:
                readability_note += f"- Calidad Proselint: {metrics.proselint_score:.0%}"
                if metrics.proselint_errors:
                    readability_note += f" ({len(metrics.proselint_errors)} sugerencias)"
                readability_note += "\n"

            # Only add note if text meets Carver standards
            if metrics.passes_carver_standards():
                readability_note += "\nâ\234\223 Cumple estÃ¡ndares Carver de claridad y concisiÃ³n."

            adjusted_text += readability_note

        return adjusted_text

        lines = ["\n## Fundamentos EpistemolÃ³gicos\n"]
        lines.append(f"Este anÃ¡lisis emplea {len(methodological_depth.paradigms_used)} paradigmas:\n")

        for p in methodological_depth.paradigms_used[:5]:
            lines.append(f"- {p}")

        # Agregar mÃ©todos clave con sus justificaciones
        key_methods = [m for m in methodological_depth.methods if m.priority <= 3]
        if key_methods:
            lines.append("\n**MÃ©todos primarios:**\n")
            for method in key_methods[:3]:
                if method.justification:
                    lines.append(f"- *{method.method_name}*: {method.justification[:150]}...")

        return "\n".join(lines)

    @classmethod
    def render_limitations_section(
        cls,
        methodological_depth: MethodologicalDepth,
    ) -> str:
        """
        Sintetiza respuesta doctoral-Carver.

        Args:
            evidence:  Evidencia ensamblada (dict con "elements", etc.)
            contract: Contrato v3 completo

        Returns:
            Respuesta en markdown, estilo Carver
        """
        # 1. Interpret contract
        dimension = self.interpreter.extract_dimension(contract)
        expected_elements = self.interpreter.extract_expected_elements(contract)
        question_intent = self.interpreter.extract_question_intent(contract)
        method_meta = self.interpreter.extract_method_metadata(contract)
        methodological_depth = self.interpreter.extract_methodological_depth(contract)

        # 2. Get dimension strategy
        strategy = get_dimension_strategy(dimension)

        # 3. Analyze evidence
        items = self.analyzer.extract_items(evidence)
        found_counts = self.analyzer.count_by_type(items)
        corroborations = self.analyzer.find_corroborations(items)
        contradictions = self.analyzer.find_contradictions(items)

        # 4. Identify gaps
        gaps = self.gap_analyzer.identify_gaps(expected_elements, found_counts, dimension)

        # 5. Compute bayesian confidence
        confidence = self.confidence_engine.compute(
            items, gaps, corroborations, contradictions
        )

        # 6. Render components
        verdict = self.renderer.render_verdict(strategy, gaps, items)
        evidence_stmts = self.renderer.render_evidence_statements(items, found_counts)
        gap_stmts = self.renderer.render_gap_statements(gaps)
        conf_stmt = self.renderer.render_confidence_statement(confidence, strategy)
        method_note = self.renderer.render_method_note(method_meta)

        # 7. Compose answer
        answer = CarverAnswer(
            verdict=verdict,
            evidence_statements=evidence_stmts,
            gap_statements=gap_stmts,
            confidence_result=confidence,
            confidence_statement=conf_stmt,
            question_text=question_intent["question_text"],
            dimension=dimension,
            method_note=method_note,
            methodological_depth=methodological_depth,
            synthesis_trace={
                "dimension": dimension.value,
                "items_count": len(items),
                "gaps_count": len(gaps),
                "critical_gaps":  sum(1 for g in gaps if g.severity == GapSeverity.CRITICAL),
                "corroborations":  len(corroborations),
                "contradictions": len(contradictions),
                "confidence": confidence.point_estimate,
            }
        )

        # 8. Render final output
        return self.renderer.render_full_answer(answer)

    def synthesize_structured(
        self,
        evidence: Dict[str, Any],
        contract: Dict[str, Any],
    ) -> CarverAnswer:
        """
        Returns structured CarverAnswer instead of string.

        Useful for further processing or integration.
        """
        # Same logic as synthesize but returns answer object
        dimension = self.interpreter.extract_dimension(contract)
        expected_elements = self.interpreter.extract_expected_elements(contract)
        question_intent = self.interpreter.extract_question_intent(contract)
        method_meta = self.interpreter.extract_method_metadata(contract)
        methodological_depth = self.interpreter.extract_methodological_depth(contract)

        strategy = get_dimension_strategy(dimension)

        items = self.analyzer.extract_items(evidence)
        found_counts = self.analyzer.count_by_type(items)
        corroborations = self.analyzer.find_corroborations(items)
        contradictions = self.analyzer.find_contradictions(items)

        gaps = self.gap_analyzer.identify_gaps(expected_elements, found_counts, dimension)

        confidence = self.confidence_engine.compute(
            items, gaps, corroborations, contradictions
        )

        verdict = self.renderer.render_verdict(strategy, gaps, items)
        evidence_stmts = self.renderer.render_evidence_statements(items, found_counts)
        gap_stmts = self.renderer.render_gap_statements(gaps)
        conf_stmt = self.renderer.render_confidence_statement(confidence, strategy)
        method_note = self.renderer.render_method_note(method_meta)

        return CarverAnswer(
            verdict=verdict,
            evidence_statements=evidence_stmts,
            gap_statements=gap_stmts,
            confidence_result=confidence,
            confidence_statement=conf_stmt,
            question_text=question_intent["question_text"],
            dimension=dimension,
            method_note=method_note,
            methodological_depth=methodological_depth,
            synthesis_trace={
                "dimension": dimension.value,
                "items_count":  len(items),
                "gaps_count": len(gaps),
                "critical_gaps": sum(1 for g in gaps if g.severity == GapSeverity.CRITICAL),
                "corroborations": len(corroborations),
                "contradictions": len(contradictions),
                "confidence": confidence.point_estimate,
            }
        )

    def synthesize_macro(
        self,
        meso_results: List[Any],  # List[MesoQuestionResult]
        coverage_matrix: Optional[Dict[Tuple[str, str], float]] = None,
        macro_question_text: str = "Â¿El Plan de Desarrollo presenta una visiÃ³n integral y coherente?",
    ) -> Dict[str, Any]:
        """
        Sintetiza respuesta macro-level con anÃ¡lisis de divergencia PAÃ\227DIM.

        AgregaciÃ³n holÃstica de mÃºltiples meso-questions con:
        - AnÃ¡lisis de cobertura PAÃ\227DIM (10 policy areas Ã\227 6 dimensions)
        - IdentificaciÃ³n de divergencias crÃticas
        - CÃ¡lculo de score holÃstico calibrado
        - GeneraciÃ³n de hallazgos, fortalezas y debilidades

        Args:
            meso_results: Lista de resultados de meso-questions
            coverage_matrix: Matriz PAÃ\227DIM con scores {("PA01", "DIM01"): 0.85, ...}
            macro_question_text: Texto de la pregunta macro

        Returns:
            Dict con estructura de MacroQuestionResult:
                - score: Score holÃstico 0-1
                - scoring_level: Nivel (excelente/bueno/aceptable/insuficiente)
                - hallazgos: Lista de hallazgos globales
                - recomendaciones: Lista de recomendaciones priorizadas
                - fortalezas: Fortalezas identificadas
                - debilidades: Debilidades identificadas (gaps)
                - divergence_analysis: AnÃ¡lisis PAÃ\227DIM detallado
        """
        # 1. Analizar cobertura PAÃ\227DIM si estÃ¡ disponible
        divergence_analysis = {}
        if coverage_matrix:
            divergence_analysis = self._analyze_pa_dim_divergence(coverage_matrix)

        # 2. Agregar scores de meso-questions
        meso_scores = [m.get("score", 0.0) if isinstance(m, dict) else getattr(m, "score", 0.0)
                       for m in meso_results]

        if not meso_scores:
            base_score = 0.0
        else:
            # Promedio ponderado con penalizaciÃ³n por varianza alta
            base_score = statistics.mean(meso_scores)
            if len(meso_scores) > 1:
                variance = statistics.variance(meso_scores)
                # Penalizar inconsistencia (varianza alta)
                variance_penalty = min(0.15, variance * 0.3)
                base_score = max(0.0, base_score - variance_penalty)

        # 3. Ajustar score con anÃ¡lisis de divergencia
        if divergence_analysis:
            coverage_score = divergence_analysis.get("overall_coverage", 1.0)
            critical_gaps_count = divergence_analysis.get("critical_gaps_count", 0)

            # Penalizar gaps crÃticos en PAÃ\227DIM
            gap_penalty = min(0.25, critical_gaps_count * 0.05)

            # Score final como promedio ponderado
            final_score = (0.7 * base_score + 0.3 * coverage_score) - gap_penalty
        else:
            final_score = base_score

        final_score = max(0.0, min(1.0, final_score))

        # 4. Determinar nivel de scoring
        if final_score >= 0.85:
            scoring_level = "excelente"
        elif final_score >= 0.70:
            scoring_level = "bueno"
        elif final_score >= 0.55:
            scoring_level = "aceptable"
        else:
            scoring_level = "insuficiente"

        # 5. Generar hallazgos globales
        hallazgos = self._generate_macro_hallazgos(
            meso_results, divergence_analysis, final_score
        )

        # 6. Generar fortalezas y debilidades
        fortalezas, debilidades = self._identify_strengths_weaknesses(
            meso_results, divergence_analysis
        )

        # 7. Generar recomendaciones priorizadas
        recomendaciones = self._generate_macro_recommendations(
            debilidades, divergence_analysis
        )

        # 8. Construir resultado macro
        return {
            "score": round(final_score, 3),
            "scoring_level": scoring_level,
            "aggregation_method": "holistic_assessment",
            "meso_results": meso_results,
            "n_meso_evaluated": len(meso_results),
            "hallazgos": hallazgos,
            "recomendaciones": recomendaciones,
            "fortalezas": fortalezas,
            "debilidades": debilidades,
            "divergence_analysis": divergence_analysis,
            "metadata": {
                "question_text": macro_question_text,
                "synthesis_method": "doctoral_carver_macro_v2",
                "base_score": round(base_score, 3),
                "coverage_adjusted": coverage_matrix is not None,
            }
        }

    def _analyze_pa_dim_divergence(
        self,
        coverage_matrix: Dict[Tuple[str, str], float]
    ) -> Dict[str, Any]:
        """
        Analiza divergencia en matriz PAÃ\227DIM (10Ã\2276 = 60 cÃ©lulas).

        Identifica:
        - Cobertura global (% de cÃ©lulas con score >= threshold)
        - Gaps crÃticos (cÃ©lulas con score < 0.5)
        - PAs y DIMs con baja cobertura
        - Patrones de divergencia
        """
        if not coverage_matrix:
            return {}

        # Definir umbrales
        THRESHOLD_ACCEPTABLE = 0.55
        THRESHOLD_CRITICAL = 0.50

        # 1. AnÃ¡lisis global
        all_scores = list(coverage_matrix.values())
        if not all_scores:
            return {"overall_coverage": 0.0, "critical_gaps_count": 0}

        overall_coverage = statistics.mean(all_scores)
        cells_above_threshold = sum(1 for s in all_scores if s >= THRESHOLD_ACCEPTABLE)
        coverage_percentage = cells_above_threshold / len(all_scores) if all_scores else 0.0

        # 2. Identificar gaps crÃticos
        critical_gaps = [
            (pa, dim, score)
            for (pa, dim), score in coverage_matrix.items()
            if score < THRESHOLD_CRITICAL
        ]

        # 3. AnÃ¡lisis por Policy Area
        policy_areas = set(pa for (pa, dim) in coverage_matrix.keys())
        pa_scores = {}
        for pa in policy_areas:
            pa_cells = [score for (p, d), score in coverage_matrix.items() if p == pa]
            pa_scores[pa] = statistics.mean(pa_cells) if pa_cells else 0.0

        low_coverage_pas = [pa for pa, score in pa_scores.items() if score < THRESHOLD_ACCEPTABLE]

        # 4. AnÃ¡lisis por Dimension
        dimensions = set(dim for (pa, dim) in coverage_matrix.keys())
        dim_scores = {}
        for dim in dimensions:
            dim_cells = [score for (p, d), score in coverage_matrix.items() if d == dim]
            dim_scores[dim] = statistics.mean(dim_cells) if dim_cells else 0.0

        low_coverage_dims = [dim for dim, score in dim_scores.items() if score < THRESHOLD_ACCEPTABLE]

        # 5. Identificar patrones de divergencia
        divergence_patterns = []

        if low_coverage_pas:
            divergence_patterns.append(
                f"Ã\201reas de polÃtica con baja cobertura: {', '.join(low_coverage_pas)}"
            )

        if low_coverage_dims:
            dim_names = {
                "DIM01": "Insumos",
                "DIM02": "Actividades",
                "DIM03": "Productos",
                "DIM04": "Resultados",
                "DIM05": "Impactos",
                "DIM06": "Causalidad"
            }
            dim_labels = [dim_names.get(d, d) for d in low_coverage_dims]
            divergence_patterns.append(
                f"Dimensiones con baja cobertura: {', '.join(dim_labels)}"
            )

        if critical_gaps:
            # Agrupar por PA
            gaps_by_pa = defaultdict(int)
            for pa, dim, score in critical_gaps:
                gaps_by_pa[pa] += 1

            top_gap_pas = sorted(gaps_by_pa.items(), key=lambda x: x[1], reverse=True)[:3]
            if top_gap_pas:
                pa_list = [f"{pa} ({count} gaps)" for pa, count in top_gap_pas]
                divergence_patterns.append(
                    f"PAs con mÃ¡s gaps crÃticos: {', '.join(pa_list)}"
                )

        return {
            "overall_coverage": round(overall_coverage, 3),
            "coverage_percentage": round(coverage_percentage, 3),
            "total_cells": len(coverage_matrix),
            "cells_above_threshold": cells_above_threshold,
            "critical_gaps_count": len(critical_gaps),
            "critical_gaps": critical_gaps[:5],  # Top 5 para no sobrecargar
            "low_coverage_pas": low_coverage_pas,
            "low_coverage_dims": low_coverage_dims,
            "pa_scores": {pa: round(score, 3) for pa, score in pa_scores.items()},
            "dim_scores": {dim: round(score, 3) for dim, score in dim_scores.items()},
            "divergence_patterns": divergence_patterns,
        }

    def _generate_macro_hallazgos(
        self,
        meso_results: List[Any],
        divergence_analysis: Dict[str, Any],
        final_score: float,
    ) -> List[str]:
        """Genera hallazgos globales del anÃ¡lisis macro."""
        hallazgos = []

        # 1. Hallazgo sobre score global
        if final_score >= 0.85:
            hallazgos.append(
                "El plan presenta un nivel excelente de integraciÃ³n y coherencia global."
            )
        elif final_score >= 0.70:
            hallazgos.append(
                "El plan muestra un nivel bueno de articulaciÃ³n entre dimensiones."
            )
        elif final_score >= 0.55:
            hallazgos.append(
                "El plan alcanza un nivel aceptable de coherencia, con Ã¡reas de mejora."
            )
        else:
            hallazgos.append(
                "El plan presenta deficiencias significativas en integraciÃ³n y coherencia."
            )

        # 2. Hallazgos de meso-questions
        if meso_results:
            high_scoring_mesos = [
                m for m in meso_results
                if (isinstance(m, dict) and m.get("score", 0) >= 0.80) or
                   (hasattr(m, "score") and m.score >= 0.80)
            ]
            low_scoring_mesos = [
                m for m in meso_results
                if (isinstance(m, dict) and m.get("score", 0) < 0.55) or
                   (hasattr(m, "score") and m.score < 0.55)
            ]

            if high_scoring_mesos:
                hallazgos.append(
                    f"{len(high_scoring_mesos)} de {len(meso_results)} clusters muestran alto desempeÃ±o."
                )

            if low_scoring_mesos:
                hallazgos.append(
                    f"{len(low_scoring_mesos)} clusters requieren atenciÃ³n prioritaria."
                )

        # 3. Hallazgos de divergencia PAÃ\227DIM
        if divergence_analysis:
            coverage_pct = divergence_analysis.get("coverage_percentage", 0.0)
            critical_gaps = divergence_analysis.get("critical_gaps_count", 0)

            if coverage_pct >= 0.80:
                hallazgos.append(
                    f"Cobertura PAÃ\227DIM: {coverage_pct:.0%} de cÃ©lulas con nivel aceptable."
                )
            else:
                status = "â\234\227"
                failed += 1
            lines.append(f"- {label}: {actual}/{expected} {status}")

        total = passed + failed
        if total > 0:
            pct = int(passed / total * 100)
            lines.append(f"\n**Cumplimiento:** {pct}% ({passed}/{total})")

        return "\n".join(lines)

    @classmethod
    def render_actionable_insights(
        cls,
        methodological_depth: MethodologicalDepth,
        gaps: List[EvidenceGap],
    ) -> str:
        """v3.0: Render insights accionables basados en el contrato."""
        lines = ["\n## Insights Accionables\n"]

        # Mapear gaps a mÃ©todos relevantes
        gap_types = {g.element_type for g in gaps if g.severity in (GapSeverity.CRITICAL, GapSeverity.MAJOR)}

        relevant_insights = []
        for method in methodological_depth.methods:
            for insight in method.actionable_insights:
                # Buscar insights relevantes a los gaps encontrados
                insight_lower = insight.lower()
                if any(gt.replace("_", " ") in insight_lower for gt in gap_types):
                    relevant_insights.append((method.method_name, insight))
                elif "few" in insight_lower or "no" in insight_lower or "missing" in insight_lower:
                    relevant_insights.append((method.method_name, insight))

        if relevant_insights:
            for method_name, insight in relevant_insights[:5]:
                lines.append(f"- **{method_name}**: {insight}")
        else:
            lines.append("- Sin insights especÃficos para los gaps detectados.")

        return "\n".join(lines)

    @classmethod
    def render_theoretical_references(
        cls,
        references: List[str],
    ) -> str:
        """v3.0: Render referencias teÃ³ricas en formato acadÃ©mico."""
        if not references:
            return ""

        lines = ["\n## Referencias TeÃ³ricas\n"]
        lines.append("Fundamentos citados en la metodologÃa:\n")

        unique_refs = list(set(references))[:10]
        for ref in unique_refs:
            lines.append(f"- {ref}")

        return "\n".join(lines)

    @classmethod
    def render_full_answer(cls, answer: CarverAnswer) -> str:
        """v3.0: Render complete doctoral answer."""
        sections = []

        # Question context
        sections.append(f"**Pregunta**: {answer.question_text}\n")

        # Verdict (the core)
        sections.append(f"## Respuesta\n\n{answer.verdict}\n")

        # Evidence (facts only)
        if answer.evidence_statements:
            sections.append("## Evidencia\n")
            for stmt in answer.evidence_statements:
                sections.append(f"- {stmt}")

        # Gaps (if any)
        if answer.gap_statements:
            sections.append("\n## VacÃos\n")
            for stmt in answer.gap_statements:
                sections.append(f"- {stmt}")

        # Confidence
        sections.append(f"\n## Confianza\n\n{answer.confidence_statement}")

        # v3.0: Doctoral sections
        if answer.epistemology_section:
            sections.append(answer.epistemology_section)
        if answer.limitations_section:
            sections.append(answer.limitations_section)

        if answer.benchmark_comparison:
            sections.append(answer.benchmark_comparison)

        # Theoretical references
        if answer.theoretical_references:
            ref_section = cls.render_theoretical_references(answer.theoretical_references)
            sections.append(ref_section)

        # Method note (discrete)
        sections.append(f"\n---\n*{answer.method_note}*")

        # Join all sections
        full_text = "\n".join(sections)

        # Apply readability checking
        adjusted_text, metrics = ReadabilityChecker.enforce_carver_style(full_text)

        # Add readability report
        if metrics.flesch_reading_ease or metrics.proselint_score:
            readability_note = "\n\n---\n**MÃ©tricas de Legibilidad**:\n"
            if metrics.flesch_reading_ease:
                readability_note += f"- Flesch Reading Ease: {metrics.flesch_reading_ease:.1f} "
                readability_note += ("(FÃ¡cil)" if metrics.flesch_reading_ease >= 60 else "(DifÃcil)")
                readability_note += "\n"
            if metrics.flesch_kincaid_grade:
                readability_note += f"- Nivel Educativo: {metrics.flesch_kincaid_grade:.1f} grado\n"
            if metrics.avg_sentence_length:
                readability_note += f"- Longitud Promedio: {metrics.avg_sentence_length:.1f} palabras/oraciÃ³n\n"
            if metrics.proselint_score is not None:
                readability_note += f"- Calidad Proselint: {metrics.proselint_score:.0%}"
                if metrics.proselint_errors:
                    readability_note += f" ({len(metrics.proselint_errors)} sugerencias)"
                readability_note += "\n"

            if metrics.passes_carver_standards():
                readability_note += "\nâ\234\223 Cumple estÃ¡ndares Carver de claridad y concisiÃ³n."

            adjusted_text += readability_note

        return adjusted_text


# ============================================================================
# MAIN SYNTHESIZER v3.0 (FULL EXTRACTION)
# ============================================================================

class DoctoralCarverSynthesizer:
    """
    Sintetizador Doctoral-Carver v3.0 FULL EXTRACTION.

    Combina rigor acadÃ©mico con prosa minimalista.
    Cada afirmaciÃ³n respaldada. Cada gap recon// filepath: /Users/recovered/PycharmProjects/F.A.R.F.A.N-MECHANISTIC_POLICY_PIP
ELINE_FINAL/src/farfan_pipeline/phases/Phase_two/carver.py
"""
Doctoral-Carver Narrative Synthesizer v3.0 (FULL EXTRACTION Edition)
=====================================================================

Genera respuestas PhD-level con estilo minimalista Raymond Carver:
- PrecisiÃ³n quirÃºrgica en cada afirmaciÃ³n
- Sin adornos retÃ³ricos vacÃos
- Cada palabra respaldada por evidencia
- Honestidad brutal sobre limitaciones
- Razonamiento causal explÃcito

CAMBIO MAYOR v3.0: ExtracciÃ³n COMPLETA del contrato v3
-----------------------------------------------------
v2.1 extraÃa solo: dimension, expected_elements, question_text, method_count, top 5 methods
v3.0 extrae TODO:
- 17 mÃ©todos con epistemological_foundation completo
- theoretical_framework con referencias acadÃ©micas
- technical_approach con algoritmos y limitaciones
- output_interpretation con actionable_insights
- assembly_flow de 5 pasos
- concrete_example como benchmark
- template_variable_bindings
- validation_against_expected_elements

Fundamentos TeÃ³ricos:
- Rhetorical Structure Theory (Mann & Thompson, 1988)
- Dempster-Shafer Evidence Theory (belief functions)
- Causal Inference Framework (Pearl, 2009)
- Argument Mining (Stab & Gurevych, 2017)
- Calibrated Uncertainty Quantification (Gneiting & Raftery, 2007)

Arquitectura v3.0:
1. EnhancedContractInterpreter: Extrae TODA la semÃ¡ntica del contrato v3
2. EvidenceAnalyzer: Construye grafo causal de evidencia
3. GapAnalyzer: AnÃ¡lisis multi-dimensional de vacÃos
4. BayesianConfidenceEngine: Inferencia calibrada de confianza
5. DimensionTheory: Estrategias teÃ³ricamente fundamentadas por D1-D6
6. DoctoralRenderer: Prosa con fundamentos epistemolÃ³gicos y citas
7. MacroSynthesizer: AgregaciÃ³n holÃstica con anÃ¡lisis PAÃ\227DIM
```

Phase 2 Source Code

```
Invariantes:
[INV-001] Toda afirmaciÃ³n debe tener â\211¥1 evidencia citada
[INV-002] Gaps crÃticos siempre aparecen en respuesta
[INV-003] Confianza debe ser calibrada (no optimista)
[INV-004] Estilo Carver: oraciones cortas, verbos activos, sin adverbios
[INV-005] Macro synthesis con divergencia PAÃ\227DIM explÃcita
[INV-006] v3.0: Fundamentos epistemolÃ³gicos explÃcitos en sÃntesis doctoral
[INV-007] v3.0: Limitaciones metodolÃ³gicas como caveats honestos
[INV-008] v3.0: Referencias teÃ³ricas en formato acadÃ©mico

Author: F.A.R.F.A.N Pipeline
Version: 3.0.0-FULL-EXTRACTION
"""

from __future__ import annotations

import math
import re
import statistics
from abc import ABC, abstractmethod
from collections import defaultdict
from dataclasses import dataclass, field
from enum import Enum
from typing import (
    Any,
    Dict,
    List,
    Optional,
    Tuple,
    TypeAlias,
)

# Readability and style checking libraries
try:
    import textstat  # Flesch-Kincaid and readability metrics
except ImportError:
    textstat = None  # type: ignore

try:
    from proselint.tools import lint as proselint_check  # Style and clarity checking
except ImportError:
    proselint_check = None  # type: ignore


# ==============================================================================
# TYPE SYSTEM
# ==============================================================================

Confidence: TypeAlias = float  # [0.0, 1.0]
BeliefMass: TypeAlias = float  # Dempster-Shafer belief
PlausibilityMass: TypeAlias = float  # Dempster-Shafer plausibility


class Dimension(Enum):
    """Las 6 dimensiones causales del modelo lÃ³gico."""
    D1_INSUMOS = "DIM01"      # Inputs: recursos, datos, diagnÃ³stico
    D2_ACTIVIDADES = "DIM02"  # Activities: acciones, instrumentos
    D3_PRODUCTOS = "DIM03"    # Outputs: entregables, metas
    D4_RESULTADOS = "DIM04"   # Outcomes: cambios inmediatos
    D5_IMPACTOS = "DIM05"     # Impacts: cambios largo plazo
    D6_CAUSALIDAD = "DIM06"   # Causality: lÃ³gica, M&E, adaptaciÃ³n


class EvidenceStrength(Enum):
    """Fuerza de evidencia segÃºn jerarquÃa epistemolÃ³gica."""
    DEFINITIVE = "definitive"      # Dato oficial verificable
    STRONG = "strong"              # MÃºltiples fuentes concordantes
    MODERATE = "moderate"          # Fuente Ãºnica confiable
    WEAK = "weak"                  # Inferido o parcial
    ABSENT = "absent"              # No encontrado


class GapSeverity(Enum):
    """Severidad de gaps con implicaciones para scoring."""
    CRITICAL = "critical"     # Bloquea evaluaciÃ³n positiva
    MAJOR = "major"           # Reduce score significativamente
    MINOR = "minor"           # Nota pero no bloquea
    COSMETIC = "cosmetic"     # Mejora deseable


class ArgumentRole(Enum):
    """Roles argumentativos (RST-inspired)."""
    CLAIM = "claim"           # AfirmaciÃ³n principal
    EVIDENCE = "evidence"     # Soporte factual
    WARRANT = "warrant"       # JustificaciÃ³n del vÃnculo
    QUALIFIER = "qualifier"   # LimitaciÃ³n/condiciÃ³n
    REBUTTAL = "rebuttal"     # Contraargumento reconocido
    BACKING = "backing"       # Soporte del warrant


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

__all__ = [
    # Enums
    "Dimension",
    "EvidenceStrength",
    "GapSeverity",
    "ArgumentRole",

    # Data structures
    "ExpectedElement",
    "EvidenceItem",
    "EvidenceGap",
    "ArgumentUnit",
    "BayesianConfidenceResult",
    "CarverAnswer",

    # v3.0 Data structures
    "MethodEpistemology",
    "TechnicalApproach",
    "OutputInterpretation",
    "MethodDepthEntry",
    "MethodCombinationLogic",
    "MethodologicalDepth",

    # Components
    "ContractInterpreter",
    "EvidenceAnalyzer",
    "GapAnalyzer",
    "BayesianConfidenceEngine",
    "DimensionStrategy",
    "CarverRenderer",

    # Main class
    "DoctoralCarverSynthesizer",

    # Factory
    "get_dimension_strategy",
]



################################################################################
# FILE: phase2_d_calibration_policy.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_d_calibration_policy.py
################################################################################

"""
Module: phase2_d_calibration_policy
PHASE_LABEL: Phase 2
Sequence: D
Description: Calibration policies for quality scoring

Version: 1.0.0
Last Modified: 2025-12-20
Author: F.A.R.F.A.N Policy Pipeline
License: Proprietary

This module is part of Phase 2: Analysis & Question Execution.
All files in Phase_two/ must contain PHASE_LABEL: Phase 2.
"""
from __future__ import annotations

import logging
from dataclasses import dataclass, field
from typing import Any

logger = logging.getLogger(__name__)


@dataclass
class CalibrationParameters:
    """Calibration parameters for a specific scope (global/dimension/PA/contract)."""

    confidence_threshold: float = 0.7
    method_weights: dict[str, float] = field(default_factory=dict)
    bayesian_priors: dict[str, Any] = field(default_factory=dict)
    random_seed: int = 42
    enable_belief_propagation: bool = True
    dempster_shafer_enabled: bool = True

    def validate(self) -> None:
        """Validate calibration parameters."""
        if not 0 <= self.confidence_threshold <= 1:
            raise ValueError(
                f"confidence_threshold must be in [0, 1], got {self.confidence_threshold}"
            )
        if self.random_seed < 0:
            raise ValueError(f"random_seed must be non-negative, got {self.random_seed}")


class CalibrationPolicy:
    """Manages calibration policies for JSON contract-based execution.

    Provides hierarchical calibration:
    - Global defaults
    - Dimension overrides (D1-D6)
    - Policy area overrides (PA01-PA10)
    - Contract overrides (Q001-Q300)
    """

    def __init__(self) -> None:
        self._global_params = CalibrationParameters()
        self._dimension_params: dict[str, CalibrationParameters] = {}
        self._policy_area_params: dict[str, CalibrationParameters] = {}
        self._contract_params: dict[str, CalibrationParameters] = {}

    def get_parameters(
        self,
        question_id: str,
        dimension_id: str | None = None,
        policy_area_id: str | None = None,
    ) -> CalibrationParameters:
        """Get calibration parameters for a specific context.

        Resolution order:
        1. Contract-specific (Q{i})
        2. Policy area-specific (PA{j})
        3. Dimension-specific (DIM{k})
        4. Global defaults
        """
        # Check contract-specific
        if question_id in self._contract_params:
            return self._contract_params[question_id]

        # Check policy area-specific
        if policy_area_id and policy_area_id in self._policy_area_params:
            return self._policy_area_params[policy_area_id]

        # Check dimension-specific
        if dimension_id and dimension_id in self._dimension_params:
            return self._dimension_params[dimension_id]

        # Return global defaults
        return self._global_params

    def set_dimension_parameters(
        self, dimension_id: str, params: CalibrationParameters
    ) -> None:
        """Set calibration parameters for a specific dimension (D1-D6)."""
        params.validate()
        self._dimension_params[dimension_id] = params
        logger.info(f"Set calibration parameters for dimension {dimension_id}")

    def set_policy_area_parameters(
        self, policy_area_id: str, params: CalibrationParameters
    ) -> None:
        """Set calibration parameters for a specific policy area (PA01-PA10)."""
        params.validate()
        self._policy_area_params[policy_area_id] = params
        logger.info(f"Set calibration parameters for policy area {policy_area_id}")

    def set_contract_parameters(
        self, question_id: str, params: CalibrationParameters
    ) -> None:
        """Set calibration parameters for a specific contract (Q001-Q300)."""
        params.validate()
        self._contract_params[question_id] = params
        logger.info(f"Set calibration parameters for contract {question_id}")

    def load_from_contract(self, contract: dict[str, Any]) -> CalibrationParameters:
        """Load calibration parameters from a contract specification.

        Args:
            contract: Q{i}.v3.json contract dict

        Returns:
            CalibrationParameters extracted from contract or defaults
        """
        calibration_spec = contract.get("calibration", {})

        params = CalibrationParameters(
            confidence_threshold=calibration_spec.get("confidence_threshold", 0.7),
            method_weights=calibration_spec.get("method_weights", {}),
            bayesian_priors=calibration_spec.get("bayesian_priors", {}),
            random_seed=calibration_spec.get("random_seed", 42),
            enable_belief_propagation=calibration_spec.get(
                "enable_belief_propagation", True
            ),
            dempster_shafer_enabled=calibration_spec.get(
                "dempster_shafer_enabled", True
            ),
        )

        params.validate()
        return params


class ParametrizationManager:
    """Manages runtime parametrization for 300 JSON contract executors."""

    def __init__(self, calibration_policy: CalibrationPolicy) -> None:
        self._calibration_policy = calibration_policy

    def get_execution_parameters(
        self, contract: dict[str, Any]
    ) -> dict[str, Any]:
        """Extract execution parameters from contract for executor.

        Returns dict suitable for passing to GenericContractExecutor.
        """
        identity = contract.get("identity", {})
        question_id = identity.get("question_id")
        dimension_id = identity.get("dimension_id")
        policy_area_id = identity.get("policy_area_id")

        # Get calibration parameters
        calib_params = self._calibration_policy.get_parameters(
            question_id=question_id,
            dimension_id=dimension_id,
            policy_area_id=policy_area_id,
        )

        # Build execution parameters
        return {
            "question_id": question_id,
            "dimension_id": dimension_id,
            "policy_area_id": policy_area_id,
            "calibration": {
                "confidence_threshold": calib_params.confidence_threshold,
                "method_weights": calib_params.method_weights,
                "random_seed": calib_params.random_seed,
                "enable_belief_propagation": calib_params.enable_belief_propagation,
                "dempster_shafer_enabled": calib_params.dempster_shafer_enabled,
            },
            "method_binding": contract.get("method_binding", {}),
            "evidence_assembly": contract.get("evidence_assembly", {}),
        }


class ConfidenceCalibrator:
    """Bayesian confidence calibration for multi-method outputs.

    Implements Dempster-Shafer belief propagation and calibrated
    confidence intervals for method aggregation.
    """

    def __init__(self, calibration_policy: CalibrationPolicy) -> None:
        self._calibration_policy = calibration_policy

    def calibrate_confidence(
        self,
        method_outputs: list[dict[str, Any]],
        question_id: str,
        dimension_id: str,
        policy_area_id: str,
    ) -> float:
        """Calibrate overall confidence from multi-method outputs.

        Uses Bayesian aggregation with method-specific weights.

        Returns:
            Calibrated confidence score in [0, 1]
        """
        params = self._calibration_policy.get_parameters(
            question_id=question_id,
            dimension_id=dimension_id,
            policy_area_id=policy_area_id,
        )

        if not method_outputs:
            return 0.0

        # Extract confidence scores from method outputs
        confidences = []
        weights = []

        for output in method_outputs:
            conf = output.get("confidence", 0.5)
            method_name = output.get("method_name", "unknown")
            weight = params.method_weights.get(method_name, 1.0)

            confidences.append(conf)
            weights.append(weight)

        # Weighted average
        if sum(weights) == 0:
            return 0.0

        calibrated = sum(c * w for c, w in zip(confidences, weights)) / sum(weights)

        # Apply Dempster-Shafer if enabled
        if params.dempster_shafer_enabled:
            calibrated = self._apply_dempster_shafer(calibrated, method_outputs)

        return min(max(calibrated, 0.0), 1.0)

    def _apply_dempster_shafer(
        self, base_confidence: float, method_outputs: list[dict[str, Any]]
    ) -> float:
        """Apply Dempster-Shafer belief propagation.
        This is a simplified implementation. Full Dempster-Shafer
        is implemented in EvidenceNexus.
        """
        # Simplified: adjust confidence based on method agreement
        if len(method_outputs) < 2:
            return base_confidence

        # Calculate variance in method confidences
        confidences = [o.get("confidence", 0.5) for o in method_outputs]
        variance = sum((c - base_confidence) ** 2 for c in confidences) / len(
            confidences
        )

        # Reduce confidence if high variance (methods disagree)
        disagreement_penalty = min(variance * 2, 0.3)

        return base_confidence * (1 - disagreement_penalty)



################################################################################
# FILE: phase2_e_contract_validator_cqvr.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_e_contract_validator_cqvr.py
################################################################################

"""Contract Validator using CQVR (Calibration, Quality, Validation, Reliability) framework.

PHASE_LABEL: Phase 2
PHASE_COMPONENT: CQVR Validator
PHASE_ROLE: Validates executor contracts using multi-tier quality scoring

from __future__ import annotations

import re
from dataclasses import dataclass, field
from enum import Enum
from typing import Any


class TriageDecision(Enum):
    PRODUCCION = "PRODUCCION"
    REFORMULAR = "REFORMULAR"
    PARCHEAR = "PARCHEAR"


@dataclass
class CQVRScore:
    tier1_score: float
    tier2_score: float
    tier3_score: float
    total_score: float
    tier1_max: float = 55.0
    tier2_max: float = 30.0
    tier3_max: float = 15.0
    total_max: float = 100.0
    component_scores: dict[str, float] = field(default_factory=dict)
    component_details: dict[str, dict[str, Any]] = field(default_factory=dict)

    @property
    def tier1_percentage(self) -> float:
        return (self.tier1_score / self.tier1_max) * 100 if self.tier1_max > 0 else 0

    @property
    def tier2_percentage(self) -> float:
        return (self.tier2_score / self.tier2_max) * 100 if self.tier2_max > 0 else 0

    @property
    def tier3_percentage(self) -> float:
        return (self.tier3_score / self.tier3_max) * 100 if self.tier3_max > 0 else 0

    @property
    def total_percentage(self) -> float:
        return (self.total_score / self.total_max) * 100 if self.total_max > 0 else 0


@dataclass
class ContractTriageDecision:
    decision: TriageDecision
    score: CQVRScore
    blockers: list[str]
    warnings: list[str]
    recommendations: list[dict[str, Any]]
    rationale: str

    def is_production_ready(self) -> bool:
        return self.decision == TriageDecision.PRODUCCION

    def requires_reformulation(self) -> bool:
        return self.decision == TriageDecision.REFORMULAR

    def can_be_patched(self) -> bool:
        return self.decision == TriageDecision.PARCHEAR


class CQVRValidator:
```

Phase 2 Source Code

```
    TIER1_THRESHOLD = 35.0
    TIER1_PRODUCTION_THRESHOLD = 45.0
    TOTAL_PRODUCTION_THRESHOLD = 80.0

    def __init__(self) -> None:
        self.blockers: list[str] = []
        self.warnings: list[str] = []
        self.recommendations: list[dict[str, Any]] = []

    def validate_contract(self, contract: dict[str, Any]) -> ContractTriageDecision:
        self.blockers = []
        self.warnings = []
        self.recommendations = []

        tier1_score = self._evaluate_tier1(contract)
        tier2_score = self._evaluate_tier2(contract)
        tier3_score = self._evaluate_tier3(contract)

        score = CQVRScore(
            tier1_score=tier1_score,
            tier2_score=tier2_score,
            tier3_score=tier3_score,
            total_score=tier1_score + tier2_score + tier3_score
        )

        decision = self._make_triage_decision(score)
        rationale = self._generate_rationale(decision, score)

        return ContractTriageDecision(
            decision=decision,
            score=score,
            blockers=self.blockers.copy(),
            warnings=self.warnings.copy(),
            recommendations=self.recommendations.copy(),
            rationale=rationale
        )

    def _evaluate_tier1(self, contract: dict[str, Any]) -> float:
        a1_score = self.verify_identity_schema_coherence(contract)
        a2_score = self.verify_method_assembly_alignment(contract)
        a3_score = self.verify_signal_requirements(contract)
        a4_score = self.verify_output_schema(contract)

        return a1_score + a2_score + a3_score + a4_score

    def _evaluate_tier2(self, contract: dict[str, Any]) -> float:
        b1_score = self.verify_pattern_coverage(contract)
        b2_score = self.verify_method_specificity(contract)
        b3_score = self.verify_validation_rules(contract)

        return b1_score + b2_score + b3_score

    def _evaluate_tier3(self, contract: dict[str, Any]) -> float:
        c1_score = self.verify_documentation_quality(contract)
        c2_score = self.verify_human_template(contract)
        c3_score = self.verify_metadata_completeness(contract)

        return c1_score + c2_score + c3_score

    def verify_identity_schema_coherence(self, contract: dict[str, Any]) -> float:
        identity = contract.get("identity", {})
        output_schema = contract.get("output_contract", {}).get("schema", {})
        properties = output_schema.get("properties", {})

        score = 0.0
        max_score = 20.0

        fields_to_check = {
            "question_id": 5.0,
            "policy_area_id": 5.0,
            "dimension_id": 5.0,
            "question_global": 3.0,
            "base_slot": 2.0
        }

        for field, points in fields_to_check.items():
            identity_value = identity.get(field)
            schema_prop = properties.get(field, {})
            schema_value = schema_prop.get("const")

            if identity_value is not None and schema_value is not None:
                if identity_value == schema_value:
                    score += points
                else:
                    self.blockers.append(
                        f"A1: Identity-Schema mismatch for '{field}': "
                        f"identity={identity_value}, schema={schema_value}"
                    )
            else:
                if identity_value is None:
                    self.blockers.append(f"A1: Missing '{field}' in identity")
                if schema_value is None:
                    self.warnings.append(f"A1: Missing const for '{field}' in output_schema")

        return score
```

Phase 2 Source Code

```
    def verify_method_assembly_alignment(self, contract: dict[str, Any]) -> float:
        method_binding = contract.get("method_binding", {})
        methods = method_binding.get("methods", [])
        evidence_assembly = contract.get("evidence_assembly", {})
        assembly_rules = evidence_assembly.get("assembly_rules", [])

        score = 0.0
        max_score = 20.0

        if not methods:
            self.blockers.append("A2: No methods defined in method_binding")
            return 0.0

        provides_set = set()
        for method in methods:
            provides = method.get("provides", "")
            if provides:
                provides_set.add(provides)

        method_count_declared = method_binding.get("method_count", len(methods))
        if method_count_declared == len(methods):
            score += 3.0
        else:
            self.warnings.append(
                f"A2: method_count mismatch: "
                f"declared={method_count_declared}, actual={len(methods)}"
            )

        sources_referenced = set()
        orphan_sources = []

        for rule in assembly_rules:
            sources = rule.get("sources", [])
            for source in sources:
                if isinstance(source, dict):
                    source_key = source.get("namespace", "")
                    if source_key and not source_key.startswith("*."):
                        sources_referenced.add(source_key)
                        if source_key not in provides_set:
                            orphan_sources.append(source_key)
                elif isinstance(source, str):
                    if not source.startswith("*."):
                        sources_referenced.add(source)
                        if source not in provides_set:
                            orphan_sources.append(source)

        if not orphan_sources:
            score += 10.0
        else:
            self.blockers.append(
                f"A2: Assembly sources not in provides: {orphan_sources[:5]}"
            )

        usage_ratio = len(sources_referenced) / len(provides_set) if provides_set else 0
        if usage_ratio >= 0.9:
            score += 5.0
        elif usage_ratio >= 0.7:
            score += 3.0
        elif usage_ratio >= 0.5:
            score += 1.0
        else:
            self.warnings.append(
                f"A2: Low method usage ratio: {usage_ratio:.1%} "
                f"({len(sources_referenced)}/{len(provides_set)})"
            )

        if not orphan_sources:
            score += 2.0

        return score

    def verify_signal_requirements(self, contract: dict[str, Any]) -> float:
        signal_requirements = contract.get("signal_requirements", {})

        score = 0.0
        max_score = 10.0

        mandatory_signals = signal_requirements.get("mandatory_signals", [])
        threshold = signal_requirements.get("minimum_signal_threshold", 0.0)
        aggregation = signal_requirements.get("signal_aggregation", "")

        if mandatory_signals and threshold <= 0:
            self.blockers.append(
                f"A3: CRITICAL - minimum_signal_threshold={threshold} "
                "but mandatory_signals defined. "
                "This allows zero-strength signals to pass validation."
            )
            return 0.0

        if mandatory_signals and threshold > 0:
            score += 5.0
        elif not mandatory_signals:
            score += 5.0

        if mandatory_signals and all(isinstance(s, str) for s in mandatory_signals):
            score += 3.0
        elif mandatory_signals:
            self.warnings.append("A3: Some mandatory_signals are not well-formed strings")

        if aggregation in ["weighted_mean", "minimum", "product", "harmonic_mean"]:
            score += 2.0
        elif aggregation:
            self.warnings.append(f"A3: Unknown signal_aggregation method: {aggregation}")

        return score

    def verify_output_schema(self, contract: dict[str, Any]) -> float:
        output_contract = contract.get("output_contract", {})
        schema = output_contract.get("schema", {})

        score = 0.0
        max_score = 5.0

        required = schema.get("required", [])
        properties = schema.get("properties", {})

        if not required:
            self.warnings.append("A4: No required fields in output_schema")
            return 0.0

        all_defined = all(field in properties for field in required)
        if all_defined:
            score += 3.0
        else:
            missing = [f for f in required if f not in properties]
            self.blockers.append(f"A4: Required fields not in properties: {missing}")

        traceability = contract.get("traceability", {})
        source_hash = traceability.get("source_hash", "")
        if source_hash and not source_hash.startswith("TODO"):
            score += 2.0
        else:
            self.warnings.append("A4: source_hash is placeholder or missing")
            score += 1.0

        return score

    def verify_pattern_coverage(self, contract: dict[str, Any]) -> float:
        question_context = contract.get("question_context", {})
        patterns = question_context.get("patterns", [])
        expected_elements = question_context.get("expected_elements", [])

        score = 0.0
        max_score = 10.0

        if not expected_elements:
            self.warnings.append("B1: No expected_elements defined")
            return 0.0

        if not patterns:
            self.warnings.append("B1: No patterns defined")
            return 0.0

        required_elements = [e for e in expected_elements if e.get("required")]

        pattern_categories = set()
        for pattern in patterns:
            if isinstance(pattern, dict):
                category = pattern.get("category", "GENERAL")
                pattern_categories.add(category)

        coverage_score = min(len(patterns) / max(len(required_elements), 1) * 5.0, 5.0)
        score += coverage_score

        confidence_weights = [
            p.get("confidence_weight", 0) for p in patterns if isinstance(p, dict)
        ]
        if confidence_weights:
            valid_weights = all(0 <= w <= 1 for w in confidence_weights)
            if valid_weights:
                score += 3.0
            else:
                self.warnings.append("B1: Some confidence_weights out of [0,1] range")

        pattern_ids = [p.get("id", "") for p in patterns if isinstance(p, dict)]
        unique_ids = len(set(pattern_ids)) == len(pattern_ids)
        if unique_ids and all(pattern_ids):
            score += 2.0
        else:
            self.warnings.append("B1: Pattern IDs not unique or missing")

        return score

    def verify_method_specificity(self, contract: dict[str, Any]) -> float:
        methodological_depth = contract.get("methodological_depth", {})
        methods = methodological_depth.get("methods", [])

        score = 0.0
        max_score = 10.0

        if not methods:
            self.warnings.append("B2: No methodological_depth.methods defined")
            return 0.0

        generic_patterns = [
            "Execute", "Process results", "Return structured output",
            "O(n) where n=input size", "Input data is preprocessed and valid"
        ]

        specific_count = 0
        boilerplate_count = 0

        for method_info in methods:
            technical = method_info.get("technical_approach", {})
            steps = technical.get("steps", [])
            complexity = technical.get("complexity", "")
            assumptions = technical.get("assumptions", [])

            is_specific = True

            for step in steps:
                step_desc = step.get("description", "")
                if any(pattern in step_desc for pattern in generic_patterns):
                    is_specific = False
                    boilerplate_count += 1
                    break

            if is_specific and complexity and not any(p in complexity for p in generic_patterns):
                specific_count += 1

        if methods:
            specificity_ratio = specific_count / len(methods)
            score += specificity_ratio * 6.0

        complexity_count = sum(
            1 for m in methods
            if m.get("technical_approach", {}).get("complexity")
            and "input size" not in m.get("technical_approach", {}).get("complexity", "")
        )
        if methods:
            score += (complexity_count / len(methods)) * 2.0

        assumptions_count = sum(
            1 for m in methods
            if m.get("technical_approach", {}).get("assumptions")
            and not any(
                "preprocessed" in str(a).lower()
                for a in m.get("technical_approach", {}).get("assumptions", [])
            )
        )
        if methods:
            score += (assumptions_count / len(methods)) * 2.0

        if boilerplate_count > len(methods) * 0.5:
            self.warnings.append(
                f"B2: High boilerplate ratio: {boilerplate_count}/{len(methods)} methods"
            )

        return score

    def verify_validation_rules(self, contract: dict[str, Any]) -> float:
        validation_rules = contract.get("validation_rules", {})
        rules = validation_rules.get("rules", [])
        question_context = contract.get("question_context", {})
        expected_elements = question_context.get("expected_elements", [])

        score = 0.0
        max_score = 10.0

        if not rules:
            self.blockers.append("B3: No validation_rules.rules defined")
            return 0.0

        required_elements = {e.get("type") for e in expected_elements if e.get("required")}

        must_contain_elements = set()
        should_contain_elements = set()

        for rule in rules:
            must_contain = rule.get("must_contain", {})
            if isinstance(must_contain, dict):
                elements = must_contain.get("elements", [])
                must_contain_elements.update(elements)

            should_contain = rule.get("should_contain", [])
            if isinstance(should_contain, list):
                for item in should_contain:
                    if isinstance(item, dict):
                        elements = item.get("elements", [])
                        should_contain_elements.update(elements)

        all_validation_elements = must_contain_elements | should_contain_elements

        if required_elements and required_elements.issubset(all_validation_elements):
            score += 5.0
        elif required_elements:
            missing = required_elements - all_validation_elements
            self.warnings.append(
                f"B3: Required elements not in validation rules: {missing}"
            )

        if must_contain_elements and should_contain_elements:
            score += 3.0
        elif must_contain_elements or should_contain_elements:
            score += 1.0

        error_handling = contract.get("error_handling", {})
        failure_contract = error_handling.get("failure_contract", {})
        if failure_contract.get("emit_code"):
            score += 2.0
        else:
            self.warnings.append("B3: No emit_code in failure_contract")

        return score

    def verify_documentation_quality(self, contract: dict[str, Any]) -> float:
        methodological_depth = contract.get("methodological_depth", {})
        methods = methodological_depth.get("methods", [])

        score = 0.0
        max_score = 5.0

        if not methods:
            self.warnings.append("C1: No methodological_depth for documentation check")
            return 0.0

        boilerplate_patterns = [
            "analytical paradigm",
            "This method contributes",
            "method implements structured analysis"
        ]

        specific_paradigms = 0
        for method_info in methods:
            epist = method_info.get("epistemological_foundation", {})
            paradigm = epist.get("paradigm", "")
            justification = epist.get("justification", "")
            framework = epist.get("theoretical_framework", [])

            is_specific = True
            for pattern in boilerplate_patterns:
                if pattern.lower() in paradigm.lower() or pattern.lower() in justification.lower():
                    is_specific = False
                    break

            if is_specific:
                specific_paradigms += 1

        if methods:
            paradigm_ratio = specific_paradigms / len(methods)
            score += paradigm_ratio * 2.0

        justifications_with_why = sum(
            1 for m in methods
            if (
                "why" in m.get("epistemological_foundation", {}).get("justification", "").lower()
                or "vs" in m.get("epistemological_foundation", {}).get("justification", "").lower()
                or "alternative"
                in m.get("epistemological_foundation", {}).get("justification", "").lower()
            )
        )
        if methods:
            score += (justifications_with_why / len(methods)) * 2.0

        has_references = any(
            m.get("epistemological_foundation", {}).get("theoretical_framework")
            for m in methods
        )
        if has_references:
            score += 1.0

        return score

    def verify_human_template(self, contract: dict[str, Any]) -> float:
        output_contract = contract.get("output_contract", {})
        human_readable = output_contract.get("human_readable_output", {})
        template = human_readable.get("template", {})

        score = 0.0
        max_score = 5.0

        identity = contract.get("identity", {})
        base_slot = identity.get("base_slot", "")
        question_id = identity.get("question_id", "")

        title = template.get("title", "")
        summary = template.get("summary", "")

        has_references = False
        if base_slot and base_slot in title:
            has_references = True
        if question_id and question_id in title:
            has_references = True
        if has_references:
            score += 3.0
        else:
            self.warnings.append("C2: Template title does not reference base_slot or question_id")

        placeholder_patterns = [
            r"\{.*?\}"
        ]

        has_placeholders = False
        for pattern in placeholder_patterns:
            if re.search(pattern, summary):
                has_placeholders = True
                break

        if has_placeholders:
            score += 2.0
        else:
            self.warnings.append("C2: Template summary has no dynamic placeholders")

        return score

    def verify_metadata_completeness(self, contract: dict[str, Any]) -> float:
        identity = contract.get("identity", {})
        traceability = contract.get("traceability", {})

        score = 0.0
        max_score = 5.0

        contract_hash = identity.get("contract_hash", "")
        if contract_hash and len(contract_hash) == 64:
            score += 2.0
        else:
            self.warnings.append("C3: contract_hash missing or invalid")

        created_at = identity.get("created_at", "")
        if created_at and "T" in created_at:
            score += 1.0
        else:
            self.warnings.append("C3: created_at missing or invalid ISO 8601 format")

        validated_against = identity.get("validated_against_schema", "")
        if validated_against:
            score += 1.0

        contract_version = identity.get("contract_version", "")
        if contract_version and "." in contract_version:
            score += 1.0

        source_hash = traceability.get("source_hash", "")
        if source_hash and not source_hash.startswith("TODO"):
            score += 3.0
        else:
            self.warnings.append("C3: source_hash is placeholder - breaks provenance chain")
            self.recommendations.append({
                "component": "C3",
                "priority": "HIGH",
                "issue": "Missing source_hash",
                "fix": (
                    "Calculate SHA256 of questionnaire_monolith.json "
                    "and update traceability.source_hash"
                ),
                "impact": "+3 pts"
            })

        return min(score, max_score)

    def _make_triage_decision(self, score: CQVRScore) -> TriageDecision:
        if score.tier1_score < self.TIER1_THRESHOLD:
            return TriageDecision.REFORMULAR

        if (score.tier1_score >= self.TIER1_PRODUCTION_THRESHOLD and
            score.total_score >= self.TOTAL_PRODUCTION_THRESHOLD):
            return TriageDecision.PRODUCCION

        if len(self.blockers) == 0 and score.total_score >= 70:
            return TriageDecision.PARCHEAR

        if len(self.blockers) <= 2 and score.tier1_score >= 40:
            return TriageDecision.PARCHEAR

        return TriageDecision.REFORMULAR

    def _generate_rationale(self, decision: TriageDecision, score: CQVRScore) -> str:
        if decision == TriageDecision.PRODUCCION:
            return (
                f"Contract approved for production: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Blockers: {len(self.blockers)}, Warnings: {len(self.warnings)}."
            )
        elif decision == TriageDecision.PARCHEAR:
            return (
                f"Contract can be patched: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Blockers: {len(self.blockers)} (resolvable), "
                f"Warnings: {len(self.warnings)}. "
                f"Apply recommended patches to reach production threshold."
            )
        else:
            reason_parts = []
            if score.tier1_score < self.TIER1_THRESHOLD:
                reason_parts.append(f"Tier 1 score below minimum threshold ({self.TIER1_THRESHOLD})")
            elif score.tier1_score < self.TIER1_PRODUCTION_THRESHOLD:
                reason_parts.append(f"Tier 1 score below production threshold ({self.TIER1_PRODUCTION_THRESHOLD})")
            if score.total_score < self.TOTAL_PRODUCTION_THRESHOLD:
                reason_parts.append(f"Total score below production threshold ({self.TOTAL_PRODUCTION_THRESHOLD})")
            if len(self.blockers) > 0:
                reason_parts.append(f"{len(self.blockers)} critical blocker(s)")

            reason_str = "; ".join(reason_parts) if reason_parts else "Failed decision criteria"

            return (
                f"Contract requires reformulation: "
                f"Tier 1: {score.tier1_score:.1f}/{score.tier1_max} "
                f"({score.tier1_percentage:.1f}%), "
                f"Total: {score.total_score:.1f}/{score.total_max} "
                f"({score.total_percentage:.1f}%). "
                f"Reasons: {reason_str}. "
                f"Contract needs substantial rework."
            )



################################################################################
# FILE: phase2_f_evidence_nexus.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_f_evidence_nexus.py
################################################################################

"""
EvidenceNexus:  Unified SOTA Evidence-to-Answer Engine

PHASE_LABEL: Phase 2
======================================================

REPLACES:
- evidence_assembler.py (merge strategies â\206\222 causal graph construction)
- evidence_validator.py (rule validation â\206\222 probabilistic consistency)
- evidence_registry.py (JSONL storage â\206\222 embedded vector store + hash chain)

ARCHITECTURE: Graph-Native Evidence Reasoning
---------------------------------------------
1.Evidence Ingestion â\206\222 Typed nodes in causal graph
2.Relationship Inference â\206\222 Edge weights via Bayesian inference
3.Consistency Validation â\206\222 Graph-theoretic conflict detection
4.Narrative Synthesis â\206\222 LLM-free template-driven answer generation
5.Provenance Tracking â\206\222 Merkle DAG with content-addressable storage

THEORETICAL FOUNDATIONS:
- Pearl’s Causal Inference (do-calculus for counterfactual reasoning)
- Dempster-Shafer Theory (belief functions for uncertainty)
- Rhetorical Structure Theory (discourse coherence)
- Information-Theoretic Validation (mutual information for relevance)

INVARIANTS:
[INV-001] All evidence nodes must have SHA-256 content hash
[INV-002] Graph must be acyclic for causal reasoning
[INV-003] Narrative must cite â\211¥1 evidence node per claim
[INV-004] Confidence intervals must be calibrated (coverage â\211¥ 0.95)
[INV-005] Hash chain must be append-only and verifiable

Author: F.A.R.F.A.N Pipeline
Version: 1.0.0
Date: 2025-12-10
"""

from __future__ import annotations

import hashlib
import json
import math
import re
import statistics
import time
from collections import defaultdict
from dataclasses import dataclass, field
from enum import Enum
from pathlib import Path
from typing import (
    Any,
    Protocol,
    Sequence,
    TypeAlias,
)

try:
    import structlog
    logger = structlog.get_logger(__name__)
except ImportError:
    import logging
    logger = logging.getLogger(__name__)


# ============================================================================
# TYPE SYSTEM
# ============================================================================

EvidenceID: TypeAlias = str  # SHA-256 hex digest
NodeID: TypeAlias = str
EdgeID: TypeAlias = str
Confidence: TypeAlias = float  # [0.0, 1.0]
BeliefMass: TypeAlias = float  # [0.0, 1.0]


class EvidenceType(Enum):
    """Taxonomy of evidence types aligned with questionnaire ontology."""
    # Quantitative
    INDICATOR_NUMERIC = "indicador_cuantitativo"
    TEMPORAL_SERIES = "serie_temporal"
    BUDGET_AMOUNT = "monto_presupuestario"
    COVERAGE_METRIC = "metrica_cobertura"
    GOAL_TARGET = "meta_cuantificada"

    # Qualitative
    OFFICIAL_SOURCE = "fuente_oficial"
    TERRITORIAL_COVERAGE = "cobertura_territorial"
    INSTITUTIONAL_ACTOR = "actor_institucional"
    POLICY_INSTRUMENT = "instrumento_politica"
    NORMATIVE_REFERENCE = "referencia_normativa"

    # Relational
    CAUSAL_LINK = "vinculo_causal"
    TEMPORAL_DEPENDENCY = "dependencia_temporal"
    CONTRADICTION = "contradiccion"
    CORROBORATION = "corroboracion"

    # Meta
    METHOD_OUTPUT = "salida_metodo"
    AGGREGATED = "agregado"
    SYNTHESIZED = "sintetizado"


class RelationType(Enum):
    """Edge types in evidence graph."""
    SUPPORTS = "supports"           # A provides evidence for B
    CONTRADICTS = "contradicts"     # A conflicts with B
    CAUSES = "causes"               # A is causal antecedent of B
    CORRELATES = "correlates"       # A and B co-occur without causation
    TEMPORALLY_PRECEDES = "precedes"  # A happens before B
    DERIVES_FROM = "derives"        # A was computed from B
    CITES = "cites"                 # A references B as source
    AGGREGATES = "aggregates"       # A is aggregation of B1... Bn


class ValidationSeverity(Enum):
    """Severity levels for validation findings."""
    CRITICAL = "critical"      # Blocks answer generation
    ERROR = "error"            # Degrades confidence significantly
    WARNING = "warning"        # Notes potential issue
    INFO = "info"              # Informational only


class AnswerCompleteness(Enum):
    """Classification of answer completeness."""
    COMPLETE = "complete"
    PARTIAL = "partial"
    INSUFFICIENT = "insufficient"
    NOT_APPLICABLE = "not_applicable"


class NarrativeSection(Enum):
    """Sections in structured narrative output."""
    DIRECT_ANSWER = "direct_answer"
    EVIDENCE_SUMMARY = "evidence_summary"
    CONFIDENCE_STATEMENT = "confidence_statement"
    SUPPORTING_DETAILS = "supporting_details"
    GAPS_AND_LIMITATIONS = "gaps_and_limitations"
    RECOMMENDATIONS = "recommendations"
    METHODOLOGY_NOTE = "methodology_note"


# ============================================================================
# CORE DATA STRUCTURES
# ============================================================================

@dataclass(frozen=True, slots=True)
class EvidenceNode:
    """
    Immutable evidence node with cryptographic identity.

    Each node represents a discrete piece of evidence extracted from
    document analysis.Nodes are content-addressed via SHA-256.
    Invariants:
    - node_id == SHA-256(canonical_json(content))
    - confidence in [0.0, 1.0]
    - belief_mass in [0.0, 1.0]
    - belief_mass + uncertainty_mass <= 1.0
    """
    node_id: EvidenceID
    evidence_type: EvidenceType
    content: dict[str, Any]

    # Confidence metrics
    confidence: Confidence
    belief_mass: BeliefMass  # Dempster-Shafer belief
    uncertainty_mass: float  # Epistemic uncertainty

    # Provenance
    source_method: str
    extraction_timestamp: float
    document_location: str | None  # Page/section reference

    # Metadata
    tags: frozenset[str] = field(default_factory=frozenset)
    parent_ids: tuple[EvidenceID, ... ] = field(default_factory=tuple)

    @classmethod
    def create(
        cls,
        evidence_type: EvidenceType,
        content: dict[str, Any],
        confidence: float,
        source_method: str,
        document_location: str | None = None,
        tags:  Sequence[str] | None = None,
        parent_ids:  Sequence[EvidenceID] | None = None,
        belief_mass: float | None = None,
        uncertainty_mass: float | None = None,
    ) -> EvidenceNode:
        """Factory method with automatic ID generation."""
        # Compute content hash for identity
        canonical = cls._canonical_json(content)
        node_id = hashlib.sha256(canonical.encode()).hexdigest()

        # Default belief mass to confidence if not specified
        bm = belief_mass if belief_mass is not None else confidence
        um = uncertainty_mass if uncertainty_mass is not None else (1.0 - confidence) * 0.5

        return cls(
            node_id=node_id,
            evidence_type=evidence_type,
            content=content,
            confidence=max(0.0, min(1.0, confidence)),
            belief_mass=max(0.0, min(1.0, bm)),
            uncertainty_mass=max(0.0, min(1.0, um)),
            source_method=source_method,
            extraction_timestamp=time.time(),
            document_location=document_location,
            tags=frozenset(tags or []),
            parent_ids=tuple(parent_ids or []),
        )

    @staticmethod
    def _canonical_json(obj: Any) -> str:
        """Deterministic JSON for hashing."""
        def default_handler(o:  Any) -> Any:
            if hasattr(o, '__dict__'):
                return o.__dict__
            if isinstance(o, Enum):
                return o.value
            return str(o)

        return json.dumps(obj, sort_keys=True, separators=(',', ':'),
                         ensure_ascii=True, default=default_handler)

    def to_dict(self) -> dict[str, Any]:
        """Serialize to dictionary."""
        return {
            "node_id": self.node_id,
            "evidence_type": self.evidence_type.value,
            "content": self.content,
            "confidence": self.confidence,
            "belief_mass": self.belief_mass,
            "uncertainty_mass": self.uncertainty_mass,
            "source_method": self.source_method,
            "extraction_timestamp": self.extraction_timestamp,
            "document_location": self.document_location,
            "tags": list(self.tags),
            "parent_ids": list(self.parent_ids),
        }


@dataclass(frozen=True, slots=True)
class EvidenceEdge:
    """
    Directed edge in evidence graph with Bayesian weight.

    Edges represent relationships between evidence nodes.
    Weight represents conditional probability P(target | source).
    """
    edge_id: EdgeID
    source_id: EvidenceID
    target_id: EvidenceID
    relation_type: RelationType
    weight:  float  # Conditional probability or strength
    confidence: Confidence
    metadata: dict[str, Any] = field(default_factory=dict)

    @classmethod
    def create(
        cls,
        source_id: EvidenceID,
        target_id: EvidenceID,
        relation_type: RelationType,
        weight: float = 1.0,
        confidence: float = 0.8,
        metadata: dict[str, Any] | None = None,
    ) -> EvidenceEdge:
        """Factory with auto-generated edge ID."""
        edge_id = hashlib.sha256(
            f"{source_id}:{target_id}:{relation_type.value}". encode()
        ).hexdigest()[:16]

        return cls(
            edge_id=edge_id,
            source_id=source_id,
            target_id=target_id,
            relation_type=relation_type,
            weight=max(0.0, min(1.0, weight)),
            confidence=max(0.0, min(1.0, confidence)),
            metadata=metadata or {},
        )


@dataclass
class ValidationFinding:
    """Single validation finding with severity and remediation."""
    finding_id: str
    severity: ValidationSeverity
    code: str
    message: str
    affected_nodes: list[EvidenceID]
    remediation: str | None = None

    def to_dict(self) -> dict[str, Any]:
        return {
            "finding_id":  self.finding_id,
            "severity": self.severity.value,
            "code": self.code,
            "message": self.message,
            "affected_nodes": self.affected_nodes,
            "remediation": self.remediation,
        }


@dataclass
class ValidationReport:
    """Complete validation report with aggregated findings."""
    is_valid: bool
    findings: list[ValidationFinding]
    critical_count: int
    error_count:  int
    warning_count: int
    validation_timestamp: float
    graph_integrity: bool
    consistency_score: float  # [0.0, 1.0]

    @classmethod
    def create(cls, findings: list[ValidationFinding]) -> ValidationReport:
        """Create report with computed aggregates."""
        critical = sum(1 for f in findings if f.severity == ValidationSeverity.CRITICAL)
        errors = sum(1 for f in findings if f.severity == ValidationSeverity.ERROR)
        warnings = sum(1 for f in findings if f.severity == ValidationSeverity.WARNING)

        # Valid only if no critical findings
        is_valid = critical == 0

        # Consistency score:  penalize errors and warnings
        base_score = 1.0
        base_score -= critical * 0.5  # Critical = -50% each
        base_score -= errors * 0.1    # Error = -10% each
        base_score -= warnings * 0.02 # Warning = -2% each
        consistency_score = max(0.0, base_score)

        return cls(
            is_valid=is_valid,
            findings=findings,
            critical_count=critical,
            error_count=errors,
            warning_count=warnings,
            validation_timestamp=time.time(),
            graph_integrity=True,  # Set by graph validation
            consistency_score=consistency_score,
        )
```

Phase 2 Source Code

```
@dataclass
class Citation:
    """Evidence citation for narrative claims."""
    node_id: EvidenceID
    evidence_type: str
    value_summary: str
    confidence: float
    source_method: str
    document_reference: str | None = None

    def render(self, format_type: str = "markdown") -> str:
        """Render citation in specified format."""
        conf_pct = f"{self.confidence * 100:.0f}%"
        if format_type == "markdown":
            ref = f" (p.{self.document_reference})" if self.document_reference else ""
            return f"[{self.evidence_type}:  {self.value_summary}]{ref} (confianza: {conf_pct})"
        return f"{self.evidence_type}: {self.value_summary} ({conf_pct})"


@dataclass
class NarrativeBlock:
    """Single block in structured narrative."""
    section: NarrativeSection
    content: str
    citations: list[Citation]
    confidence: float

    def render(self, format_type: str = "markdown") -> str:
        """Render block with citations."""
        if format_type == "markdown":
            header_map = {
                NarrativeSection.DIRECT_ANSWER:  "## Respuesta",
                NarrativeSection.EVIDENCE_SUMMARY: "### Resumen de Evidencia",
                NarrativeSection.CONFIDENCE_STATEMENT: "### Nivel de Confianza",
                NarrativeSection.SUPPORTING_DETAILS: "### AnÃ¡lisis Detallado",
                NarrativeSection.GAPS_AND_LIMITATIONS: "### Limitaciones Identificadas",
                NarrativeSection.RECOMMENDATIONS: "### Recomendaciones",
                NarrativeSection.METHODOLOGY_NOTE: "### Nota MetodolÃ³gica",
            }
            header = header_map.get(self.section, f"### {self.section.value}")
            return f"{header}\n\n{self.content}"
        return f"{self.section.value.upper()}\n{self.content}"


@dataclass
class SynthesizedAnswer:
    """Complete synthesized answer with full provenance."""
    # Core answer
    direct_answer: str
    narrative_blocks: list[NarrativeBlock]

    # Quality metrics
    completeness: AnswerCompleteness
    overall_confidence: float
    calibrated_interval: tuple[float, float]  # 95% CI

    # Evidence linkage
    primary_citations: list[Citation]
    supporting_citations: list[Citation]

    # Gaps and issues
    gaps:  list[str]
    unresolved_contradictions: list[str]

    # Provenance
    evidence_graph_hash: str
    synthesis_timestamp: float
    question_id: str

    # Trace
    synthesis_trace: dict[str, Any]

    def to_human_readable(self, format_type: str = "markdown") -> str:
        """Generate final human-readable output."""
        sections = []

        for block in self.narrative_blocks:
            sections.append(block.render(format_type))

        separator = "\n\n" if format_type == "markdown" else "\n\n"
        return separator.join(sections)

    def to_dict(self) -> dict[str, Any]:
        """Full serialization."""
        return {
            "direct_answer": self.direct_answer,
            "completeness": self.completeness.value,
            "overall_confidence":  self.overall_confidence,
            "calibrated_interval": list(self.calibrated_interval),
            "gaps": self.gaps,
            "unresolved_contradictions": self.unresolved_contradictions,
            "evidence_graph_hash": self.evidence_graph_hash,
            "synthesis_timestamp":  self.synthesis_timestamp,
            "question_id": self.question_id,
            "primary_citation_count": len(self.primary_citations),
            "supporting_citation_count": len(self.supporting_citations),
        }


# ============================================================================
# EVIDENCE GRAPH
# ============================================================================

class EvidenceGraph:
    """
    Directed acyclic graph of evidence with causal reasoning support.

    Implements:
    - Content-addressable node storage
    - Relationship inference
    - Causal path analysis
    - Conflict detection
    - Belief propagation (Dempster-Shafer)
    """

    __slots__ = (
        '_nodes', '_edges', '_adjacency', '_reverse_adjacency',
        '_type_index', '_source_index', '_hash_chain', '_last_hash',
    )

    def __init__(self) -> None:
        self._nodes: dict[EvidenceID, EvidenceNode] = {}
        self._edges: dict[EdgeID, EvidenceEdge] = {}
        self._adjacency: dict[EvidenceID, list[EdgeID]] = defaultdict(list)
        self._reverse_adjacency: dict[EvidenceID, list[EdgeID]] = defaultdict(list)
        self._type_index: dict[EvidenceType, list[EvidenceID]] = defaultdict(list)
        self._source_index: dict[str, list[EvidenceID]] = defaultdict(list)
        self._hash_chain: list[str] = []
        self._last_hash: str | None = None

    # ------------------------------------------------------------------------
    # Node Operations
    # ------------------------------------------------------------------------

    def add_node(self, node:  EvidenceNode) -> EvidenceID:
        """Add node to graph with hash chain update."""
        if node.node_id in self._nodes:
            return node.node_id  # Idempotent

        self._nodes[node.node_id] = node
        self._type_index[node.evidence_type].append(node.node_id)
        self._source_index[node.source_method].append(node.node_id)

        # Update hash chain
        self._update_hash_chain(node)

        logger.debug("node_added", node_id=node.node_id[: 12],
                    evidence_type=node.evidence_type.value)

        return node.node_id

    def add_nodes(self, nodes:  Sequence[EvidenceNode]) -> list[EvidenceID]:
        """Batch add nodes."""
        return [self.add_node(n) for n in nodes]

    def get_node(self, node_id:  EvidenceID) -> EvidenceNode | None:
        """Retrieve node by ID."""
        return self._nodes.get(node_id)

    def get_nodes_by_type(self, evidence_type: EvidenceType) -> list[EvidenceNode]:
        """Get all nodes of a specific type."""
        node_ids = self._type_index.get(evidence_type, [])
        return [self._nodes[nid] for nid in node_ids if nid in self._nodes]

    def get_nodes_by_source(self, source_method: str) -> list[EvidenceNode]:
        """Get all nodes from a specific source method."""
        node_ids = self._source_index.get(source_method, [])
        return [self._nodes[nid] for nid in node_ids if nid in self._nodes]

    # ------------------------------------------------------------------------
    # Edge Operations
    # ------------------------------------------------------------------------

    def add_edge(self, edge: EvidenceEdge) -> EdgeID:
        """Add edge to graph."""
        if edge.source_id not in self._nodes or edge.target_id not in self._nodes:
            raise ValueError(
                f"Cannot add edge:  source {edge.source_id[: 12]} or "
                f"target {edge.target_id[:12]} not in graph"
            )

        if edge.edge_id in self._edges:
            return edge.edge_id  # Idempotent

        # Check for cycle (DAG invariant)
        if self._would_create_cycle(edge.source_id, edge.target_id):
            raise ValueError(
                f"Cannot add edge: would create cycle from "
                f"{edge.source_id[:12]} to {edge.target_id[:12]}"
            )
        self._edges[edge.edge_id] = edge
        self._adjacency[edge.source_id].append(edge.edge_id)
        self._reverse_adjacency[edge.target_id].append(edge.edge_id)

        return edge.edge_id

    def _would_create_cycle(self, source:  EvidenceID, target: EvidenceID) -> bool:
        """Check if adding edge sourceâ\206\222target would create a cycle."""
        # If target can reach source, adding sourceâ\206\222target creates cycle
        visited:  set[EvidenceID] = set()
        stack = [target]

        while stack:
            current = stack.pop()
            if current == source:
                return True
            if current in visited:
                continue
            visited.add(current)

            # Follow outgoing edges
            for edge_id in self._adjacency.get(current, []):
                edge = self._edges[edge_id]
                stack.append(edge.target_id)

        return False

    def get_edges_from(self, node_id: EvidenceID) -> list[EvidenceEdge]:
        """Get outgoing edges from node."""
        edge_ids = self._adjacency.get(node_id, [])
        return [self._edges[eid] for eid in edge_ids if eid in self._edges]

    def get_edges_to(self, node_id: EvidenceID) -> list[EvidenceEdge]:
        """Get incoming edges to node."""
        edge_ids = self._reverse_adjacency.get(node_id, [])
        return [self._edges[eid] for eid in edge_ids if eid in self._edges]

    def get_edges_by_type(self, relation_type: RelationType) -> list[EvidenceEdge]:
        """Get all edges of a specific type."""
        return [e for e in self._edges.values() if e.relation_type == relation_type]

    # -------------------------------------------------------------------------
    # Graph Analysis
    # -------------------------------------------------------------------------

    def find_supporting_evidence(
        self,
        node_id: EvidenceID,
        max_depth: int = 3
    ) -> list[tuple[EvidenceNode, int]]:
        """
        Find all evidence that supports a node (transitive).
        Returns (node, depth) pairs.
        """
        results:  list[tuple[EvidenceNode, int]] = []
        visited:  set[EvidenceID] = set()

        def traverse(nid: EvidenceID, depth: int) -> None:
            if depth > max_depth or nid in visited:
                return
            visited.add(nid)

            for edge in self.get_edges_to(nid):
                if edge.relation_type in (RelationType.SUPPORTS, RelationType.DERIVES_FROM):
                    source_node = self._nodes.get(edge.source_id)
                    if source_node:
                        results.append((source_node, depth))
                        traverse(edge.source_id, depth + 1)

        traverse(node_id, 1)
        return results

    def find_contradictions(self) -> list[tuple[EvidenceNode, EvidenceNode, EvidenceEdge]]:
        """Find all contradiction pairs in graph."""
        contradictions = []
        for edge in self.get_edges_by_type(RelationType.CONTRADICTS):
            source = self._nodes.get(edge.source_id)
            target = self._nodes.get(edge.target_id)
            if source and target:
                contradictions.append((source, target, edge))
        return contradictions

    def compute_belief_propagation(self) -> dict[EvidenceID, float]:
        """
        Dempster-Shafer belief propagation across graph.

        Combines evidence using Dempster’s rule of combination.
        Returns updated belief masses for each node.
        """
        beliefs:  dict[EvidenceID, float] = {}

        # Topological sort for propagation order
        sorted_nodes = self._topological_sort()

        for node_id in sorted_nodes:
            node = self._nodes[node_id]
            incoming = self.get_edges_to(node_id)

            if not incoming:
                # Root node:  use intrinsic belief
                beliefs[node_id] = node.belief_mass
            else:
                # Combine beliefs from parents using Dempster’s rule
                combined_belief = node.belief_mass

                for edge in incoming:
                    if edge.relation_type == RelationType.SUPPORTS:
                        parent_belief = beliefs.get(edge.source_id, 0.5)
                        # Dempster’s combination (simplified)
                        combined_belief = self._dempster_combine(
                            combined_belief, parent_belief * edge.weight
                        )
                    elif edge.relation_type == RelationType.CONTRADICTS:
                        parent_belief = beliefs.get(edge.source_id, 0.5)
                        # Contradiction reduces belief
                        combined_belief *= (1 - parent_belief * edge.weight * 0.5)

                beliefs[node_id] = max(0.0, min(1.0, combined_belief))

        return beliefs

    @staticmethod
    def _dempster_combine(m1: float, m2: float) -> float:
        """Dempster’s rule of combination for two belief masses."""
        # Simplified:  assume no direct conflict
        conflict = m1 * (1 - m2) * 0.1  # Small conflict factor
        normalization = 1 - conflict
        if normalization <= 0:
            return 0.5  # Maximum uncertainty

        combined = (m1 * m2) / normalization
        return max(0.0, min(1.0, combined))

    def _topological_sort(self) -> list[EvidenceID]:
        """Topological sort of nodes (Kahn’s algorithm)."""
        in_degree:  dict[EvidenceID, int] = {nid: 0 for nid in self._nodes}

        for edge in self._edges.values():
            in_degree[edge.target_id] += 1

        queue = [nid for nid, deg in in_degree.items() if deg == 0]
        result = []

        while queue:
            node_id = queue.pop(0)
            result.append(node_id)

            for edge in self.get_edges_from(node_id):
                in_degree[edge.target_id] -= 1
                if in_degree[edge.target_id] == 0:
                    queue.append(edge.target_id)

        return result

    # -------------------------------------------------------------------------
    # Hash Chain (Provenance)
    # -------------------------------------------------------------------------

    def _update_hash_chain(self, node:  EvidenceNode) -> None:
        """Append node to hash chain."""
        chain_data = {
            "node_id": node.node_id,
            "previous_hash": self._last_hash or "",
            "timestamp": node.extraction_timestamp,
        }
        entry_hash = hashlib.sha256(
            json.dumps(chain_data, sort_keys=True).encode()
        ).hexdigest()

        self._hash_chain.append(entry_hash)
        self._last_hash = entry_hash

    def verify_hash_chain(self) -> bool:
        """Verify hash chain integrity."""
        if not self._hash_chain:
            return True

        # Would need full reconstruction to verify
        # For now, check chain exists and is non-empty
        return len(self._hash_chain) == len(self._nodes)

    def get_graph_hash(self) -> str:
        """Get hash of entire graph state."""
        if self._last_hash:
            return self._last_hash
        return hashlib.sha256(b"empty_graph").hexdigest()

    # -------------------------------------------------------------------------
    # Statistics
    # -------------------------------------------------------------------------
```

Phase 2 Source Code

```
    @property
    def node_count(self) -> int:
        return len(self._nodes)

    @property
    def edge_count(self) -> int:
        return len(self._edges)

    def get_statistics(self) -> dict[str, Any]:
        """Comprehensive graph statistics."""
        type_counts = {t.value: len(ids) for t, ids in self._type_index.items()}
        source_counts = {s:  len(ids) for s, ids in self._source_index.items()}

        confidences = [n.confidence for n in self._nodes.values()]
        avg_confidence = statistics.mean(confidences) if confidences else 0.0

        edge_type_counts = defaultdict(int)
        for edge in self._edges.values():
            edge_type_counts[edge.relation_type.value] += 1

        return {
            "node_count": self.node_count,
            "edge_count": self.edge_count,
            "by_evidence_type": type_counts,
            "by_source_method": source_counts,
            "by_edge_type": dict(edge_type_counts),
            "average_confidence": avg_confidence,
            "hash_chain_length": len(self._hash_chain),
            "graph_hash": self.get_graph_hash()[: 16],
        }


# ============================================================================
# VALIDATION ENGINE
# ============================================================================

class ValidationRule(Protocol):
    """Protocol for validation rules."""

    @property
    def code(self) -> str: ...

    @property
    def severity(self) -> ValidationSeverity: ...

    def validate(self, graph: EvidenceGraph, contract: dict[str, Any]) -> list[ValidationFinding]:  ...


class RequiredElementsRule:
    """Validate that required evidence types are present."""

    code = "REQ_ELEMENTS"
    severity = ValidationSeverity.ERROR

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        expected_elements = contract.get("question_context", {}).get("expected_elements", [])

        for elem in expected_elements:
            elem_type = elem.get("type", "")
            required = elem.get("required", False)
            minimum = elem.get("minimum", 0)

            # In the monolith, expected_elements[].type includes many "context" gates
            # (e.g., coherencia_demostrada, analisis_realismo, trazabilidad_presupuestal)
            # that are NOT EvidenceType enum values. We must still validate them.
            count, node_ids = self._count_support_for_expected_element(graph, str(elem_type))

            if required and count == 0:
                findings.append(ValidationFinding(
                    finding_id=f"REQ_{elem_type}",
                    severity=ValidationSeverity.ERROR,
                    code=self.code,
                    message=f"Required element type '{elem_type}' not found in evidence",
                    affected_nodes=[],
                    remediation=f"Ensure document analysis extracts {elem_type} elements",
                ))
            elif minimum > 0 and count < minimum:
                findings.append(ValidationFinding(
                    finding_id=f"MIN_{elem_type}",
                    severity=ValidationSeverity.WARNING,
                    code="MIN_ELEMENTS",
                    message=f"Element type '{elem_type}' has {count}/{minimum} required instances",
                    affected_nodes=node_ids[:10],
                    remediation=f"Need {minimum - count} more {elem_type} elements",
                ))

        return findings

    @staticmethod
    def _count_support_for_expected_element(
        graph: EvidenceGraph,
        expected_type: str,
    ) -> tuple[int, list[str]]:
        """Count how much evidence supports an expected element type.

        Strategy:
        - Prefer exact EvidenceType matches when possible
        - Otherwise use contract-pattern evidence nodes (source_method=contract.patterns)
          and map "context" types to categories/lexical markers.
        """
        expected = (expected_type or "").strip()
        if not expected:
            return 0, []

        # 1) Direct mapping to EvidenceType (when expected uses EvidenceType vocabulary)
        try:
            ev_type = EvidenceType(expected)
            nodes = graph.get_nodes_by_type(ev_type)
            return len(nodes), [n.node_id for n in nodes]
        except ValueError:
            pass

        # 2) Build indices from contract.patterns nodes (produced in _build_graph_from_outputs)
        pattern_nodes = graph.get_nodes_by_source("contract.patterns")
        by_category: dict[str, list[EvidenceNode]] = defaultdict(list)
        for n in pattern_nodes:
            if isinstance(n.content, dict):
                cat = str(n.content.get("category") or "GENERAL").upper()
                by_category[cat].append(n)

        def _count_nodes_with_matches(nodes: list[EvidenceNode]) -> tuple[int, list[str]]:
            ids: list[str] = []
            total = 0
            for n in nodes:
                mc = 0
                if isinstance(n.content, dict):
                    try:
                        mc = int(n.content.get("match_count", 0) or 0)
                    except Exception:
                        mc = 0
                if mc > 0:
                    total += mc
                    ids.append(n.node_id)
            return total, ids

        def _contains_any(match_texts: list[str], needles: tuple[str, ...]) -> bool:
            text = " ".join(match_texts).lower()
            return any(needle in text for needle in needles)

        # Helper to extract match texts from a node
        def _node_matches(node: EvidenceNode) -> list[str]:
            if not isinstance(node.content, dict):
                return []
            raw = node.content.get("matches", [])
            if isinstance(raw, list):
                return [str(x) for x in raw if x is not None]
            return []

        # 3) Map monolith expected element types ("contexts") to evidence signals
        # NOTE: This mapping is conservative and explainable. It can be refined
        # as we formalize contextâ\206\222pattern/validation specs in SISAS.
        et = expected.lower()

        # High-signal direct category mappings
        category_map: dict[str, str] = {
            "fuentes_oficiales": "FUENTE_OFICIAL",
            "indicadores_cuantitativos": "INDICADOR",
            "series_temporales_aÃ±os": "TEMPORAL",
            "cobertura_territorial_especificada": "TERRITORIAL",
            "rezago_temporal": "TEMPORAL",
            "ruta_transmision": "CAUSAL",
            "logica_causal_explicita": "CAUSAL",
            "teoria_cambio_explicita": "CAUSAL",
            "cadena_causal_explicita": "CAUSAL",
            "mecanismo_causal_explicito": "CAUSAL",
            "unidades_medicion": "UNIDAD_MEDIDA",
            "trazabilidad_presupuestal": "INDICADOR",
        }

        if et in category_map:
            # CAUSAL is spread across CAUSAL*, so merge all keys starting with CAUSAL
            wanted = category_map[et]
            if wanted == "CAUSAL":
                causal_nodes: list[EvidenceNode] = []
                for k, nodes in by_category.items():
                    if k.startswith("CAUSAL"):
                        causal_nodes.extend(nodes)
                return _count_nodes_with_matches(causal_nodes)
            return _count_nodes_with_matches(by_category.get(wanted, []))

        # Context-style expected elements (from your list)
        if et == "completeness":
            # Treat as: any supporting evidence exists
            total_matches = 0
            ids: list[str] = []
            for nodes in by_category.values():
                c, nid = _count_nodes_with_matches(nodes)
                total_matches += c
                ids.extend(nid)
            return total_matches, ids

        if et == "horizonte_temporal":
            return _count_nodes_with_matches(by_category.get("TEMPORAL", []))

        if et in {"asignacion_explicita", "restricciones_presupuestales", "coherencia_recursos"}:
            # Budget/resource realism: indicator/unit + key lexemes in matched snippets.
            indicator_nodes = by_category.get("INDICADOR", [])
            unit_nodes = by_category.get("UNIDAD_MEDIDA", [])
            nodes = indicator_nodes + unit_nodes
            count, ids = _count_nodes_with_matches(nodes)
            if count == 0:
                return 0, []
            # Lexeme filter (keeps it honest)
            budget_lexemes = ("presupuesto", "recursos", "financi", "monto", "millones", "cop", "$")
            supported_ids: list[str] = []
            supported_count = 0
            for n in nodes:
                matches = _node_matches(n)
                if matches and _contains_any(matches, budget_lexemes):
                    try:
                        supported_count += int(n.content.get("match_count", 0) or 0)  # type: ignore[union-attr]
                    except Exception:
                        supported_count += 1
                    supported_ids.append(n.node_id)
            return (supported_count or count), (supported_ids or ids)

        if et in {"analisis_realismo", "analisis_contextual", "evidencia_comparada"}:
            # Realism/context/comparison: prefer INDICADOR/TEMPORAL/FUENTE_OFICIAL/TERRITORIAL
            nodes = (
                by_category.get("INDICADOR", [])
                + by_category.get("TEMPORAL", [])
                + by_category.get("FUENTE_OFICIAL", [])
                + by_category.get("TERRITORIAL", [])
            )
            count, ids = _count_nodes_with_matches(nodes)
            if et == "evidencia_comparada" and count > 0:
                compar_lex = ("compar", "vs", "promedio", "nacional", "departamental", "anterior")
                supported_ids = []
                supported_count = 0
                for n in nodes:
                    matches = _node_matches(n)
                    if matches and _contains_any(matches, compar_lex):
                        supported_ids.append(n.node_id)
                        try:
                            supported_count += int(n.content.get("match_count", 0) or 0)  # type: ignore[union-attr]
                        except Exception:
                            supported_count += 1
                return (supported_count or 0), supported_ids
            return count, ids

        if et in {"supuestos_identificados", "riesgos_identificados", "ciclos_aprendizaje", "enfoque_diferencial", "gobernanza"
, "poblacion_objetivo_definida", "vinculo_diagnostico_actividad"}:
            # These are often GENERAL patterns with strong lexical anchors.
            general_nodes = by_category.get("GENERAL", [])
            count, ids = _count_nodes_with_matches(general_nodes)
            if count == 0:
                return 0, []
            anchors_by_type: dict[str, tuple[str, ...]] = {
                "supuestos_identificados": ("supuesto", "asumi", "hipÃ³tesis", "premisa"),
                "riesgos_identificados": ("riesgo", "amenaza", "mitig", "conting"),
                "ciclos_aprendizaje": ("retroaliment", "aprendiz", "mejora", "ciclo", "monitoreo"),
                "enfoque_diferencial": ("enfoque diferencial", "enfoque de gÃ©nero", "enfoque Ã©tn", "interseccional"),
                "gobernanza": ("gobernanza", "coordinaciÃ³n", "articulaciÃ³n", "comitÃ©", "mesa", "instancia"),
                "poblacion_objetivo_definida": ("poblaciÃ³n objetivo", "beneficiari", "grupo meta", "focaliz"),
                "vinculo_diagnostico_actividad": ("diagnÃ³stico", "brecha", "causa", "en respuesta", "derivado"),
            }
            anchors = anchors_by_type.get(et, tuple())
            if not anchors:
                return count, ids
            supported_ids: list[str] = []
            supported_count = 0
            for n in general_nodes:
                matches = _node_matches(n)
                if matches and _contains_any(matches, anchors):
                    supported_ids.append(n.node_id)
                    try:
                        supported_count += int(n.content.get("match_count", 0) or 0)  # type: ignore[union-attr]
                    except Exception:
                        supported_count += 1
            return supported_count, supported_ids

        # Fallback: treat as "any evidence exists" but only count nodes with matches
        total_matches = 0
        ids: list[str] = []
        for nodes in by_category.values():
            c, nid = _count_nodes_with_matches(nodes)
            total_matches += c
            ids.extend(nid)
        return total_matches, ids


class ConsistencyRule:
    """Validate internal consistency of evidence."""

    code = "CONSISTENCY"
    severity = ValidationSeverity.WARNING

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        # Check for unresolved contradictions
        contradictions = graph.find_contradictions()

        for source, target, edge in contradictions:
            if edge.confidence > 0.7:  # High-confidence contradiction
                findings.append(ValidationFinding(
                    finding_id=f"CONTRA_{edge.edge_id}",
                    severity=ValidationSeverity.WARNING,
                    code=self.code,
                    message=f"Contradiction detected between evidence nodes",
                    affected_nodes=[source.node_id, target.node_id],
                    remediation="Review contradictory evidence for resolution",
                ))

        return findings


class ConfidenceThresholdRule:
    """Validate confidence thresholds."""

    code = "CONFIDENCE"
    severity = ValidationSeverity.WARNING

    def __init__(self, min_confidence: float = 0.5):
        self.min_confidence = min_confidence

    def validate(
        self,
        graph: EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        low_confidence_nodes = [
            n for n in graph._nodes.values()
            if n.confidence < self.min_confidence
        ]

        if len(low_confidence_nodes) > graph.node_count * 0.3:
            findings.append(ValidationFinding(
                finding_id="LOW_CONF_AGGREGATE",
                severity=ValidationSeverity.WARNING,
                code=self.code,
                message=f"{len(low_confidence_nodes)}/{graph.node_count} nodes have confidence below {self.min_confidence}",
                affected_nodes=[n.node_id for n in low_confidence_nodes[: 10]],
                remediation="Consider additional evidence sources or validation",
            ))

        return findings


class GraphIntegrityRule:
    """Validate graph structural integrity."""

    code = "INTEGRITY"
    severity = ValidationSeverity.CRITICAL

    def validate(
        self,
        graph: EvidenceGraph,
        contract: dict[str, Any]
    ) -> list[ValidationFinding]:
        findings = []

        # Verify hash chain
        if not graph.verify_hash_chain():
            findings.append(ValidationFinding(
                finding_id="HASH_CHAIN_INVALID",
                severity=ValidationSeverity.CRITICAL,
                code=self.code,
                message="Hash chain integrity verification failed",
                affected_nodes=[],
                remediation="Evidence chain may be corrupted; rebuild from source",
            ))

        # Check for orphan edges
        for edge in graph._edges.values():
            if edge.source_id not in graph._nodes or edge.target_id not in graph._nodes:
                findings.append(ValidationFinding(
                    finding_id=f"ORPHAN_EDGE_{edge.edge_id}",
                    severity=ValidationSeverity.ERROR,
                    code=self.code,
                    message=f"Edge references non-existent node",
                    affected_nodes=[],
                    remediation="Remove orphan edge or add missing nodes",
                ))

        return findings


class ValidationEngine:
    """
    Probabilistic validation engine for evidence graphs.

    Replaces rule-based EvidenceValidator with graph-aware validation.
    """

    def __init__(self, rules: list[ValidationRule] | None = None):
        self.rules:  list[ValidationRule] = rules or [
            RequiredElementsRule(),
            ConsistencyRule(),
            ConfidenceThresholdRule(min_confidence=0.5),
            GraphIntegrityRule(),
        ]

    def validate(
        self,
        graph:  EvidenceGraph,
        contract: dict[str, Any]
    ) -> ValidationReport:
        """Run all validation rules and produce report."""
        all_findings:  list[ValidationFinding] = []

        for rule in self.rules:
            try:
                findings = rule.validate(graph, contract)
                all_findings.extend(findings)
            except Exception as e:
                logger.error("validation_rule_failed", rule=rule.code, error=str(e))
                all_findings.append(ValidationFinding(
                    finding_id=f"RULE_ERROR_{rule.code}",
                    severity=ValidationSeverity.ERROR,
                    code="VALIDATION_ERROR",
                    message=f"Validation rule {rule.code} failed: {e}",
                    affected_nodes=[],
                ))

        report = ValidationReport.create(all_findings)
        report.graph_integrity = graph.verify_hash_chain()

        logger.info(
            "validation_complete",
            is_valid=report.is_valid,
            critical=report.critical_count,
            errors=report.error_count,
            warnings=report.warning_count,
        )

        return report


# ============================================================================
# NARRATIVE SYNTHESIZER
# ============================================================================

class NarrativeSynthesizer:
    """
    Transform evidence graph into coherent narrative answer.

    Implements Rhetorical Structure Theory for discourse coherence.
    """

    def __init__(
        self,
        citation_threshold: float = 0.6,
        max_citations_per_claim: int = 3,
    ):
        self.citation_threshold = citation_threshold
        self.max_citations_per_claim = max_citations_per_claim

    def synthesize(
        self,
        graph: EvidenceGraph,
        question_context: dict[str, Any],
        validation:  ValidationReport,
        contract: dict[str, Any],
    ) -> SynthesizedAnswer:
        """
        Synthesize complete answer from evidence graph.

        Process:
        1.Determine answer completeness from validation
        2.Select primary and supporting evidence
        3.Generate direct answer based on question type
        4.Build narrative blocks with citations
        5.Identify gaps and contradictions
        6.Compute calibrated confidence
        """
        question_global = question_context.get("question_global", "")
        question_id = question_context.get("question_id", "UNKNOWN")
        expected_elements = question_context.get("expected_elements", [])

        # 1. Determine completeness
        completeness = self._determine_completeness(graph, expected_elements, validation)

        # 2. Select evidence
        primary_nodes = self._select_primary_evidence(graph, expected_elements)
        supporting_nodes = self._select_supporting_evidence(graph, primary_nodes)

        # 3. Build citations
        primary_citations = [self._node_to_citation(n) for n in primary_nodes]
        supporting_citations = [self._node_to_citation(n) for n in supporting_nodes]

        # 4. Generate direct answer
        answer_type = self._infer_answer_type(question_global)
        direct_answer = self._generate_direct_answer(
            graph, question_global, answer_type, completeness, primary_citations
        )

        # 5. Build narrative blocks
        blocks = self._build_narrative_blocks(
            direct_answer, graph, completeness, validation,
            primary_citations, supporting_citations
        )

        # 6. Identify gaps and contradictions
        gaps = self._identify_gaps(graph, expected_elements)
        contradictions = self._format_contradictions(graph.find_contradictions())

        # 7. Compute confidence
        overall_confidence, calibrated_interval = self._compute_confidence(
            graph, validation, completeness
        )

        return SynthesizedAnswer(
            direct_answer=direct_answer,
            narrative_blocks=blocks,
            completeness=completeness,
            overall_confidence=overall_confidence,
            calibrated_interval=calibrated_interval,
            primary_citations=primary_citations,
            supporting_citations=supporting_citations,
            gaps=gaps,
            unresolved_contradictions=contradictions,
            evidence_graph_hash=graph.get_graph_hash(),
            synthesis_timestamp=time.time(),
            question_id=question_id,
            synthesis_trace={
                "answer_type": answer_type,
                "primary_evidence_count": len(primary_nodes),
                "supporting_evidence_count":  len(supporting_nodes),
                "validation_passed": validation.is_valid,
            },
        )

    def _determine_completeness(
        self,
        graph: EvidenceGraph,
        expected_elements: list[dict[str, Any]],
        validation: ValidationReport,
    ) -> AnswerCompleteness:
        """Determine answer completeness based on evidence coverage."""
        if validation.critical_count > 0:
            return AnswerCompleteness.INSUFFICIENT

        required_types = [e["type"] for e in expected_elements if e.get("required")]
        found_types = set()

        for ev_type in EvidenceType:
            if graph.get_nodes_by_type(ev_type):
                found_types.add(ev_type.value)

        missing_required = set(required_types) - found_types

        if not missing_required:
            return AnswerCompleteness.COMPLETE
        elif len(found_types) > 0:
            return AnswerCompleteness.PARTIAL
        else:
            return AnswerCompleteness.INSUFFICIENT

    def _select_primary_evidence(
        self,
        graph: EvidenceGraph,
        expected_elements: list[dict[str, Any]],
    ) -> list[EvidenceNode]:
        """Select primary evidence nodes for answer."""
        primary = []

        # Prioritize required elements
        required_types = [e["type"] for e in expected_elements if e.get("required")]

        for type_str in required_types:
            try:
                ev_type = EvidenceType(type_str)
                nodes = graph.get_nodes_by_type(ev_type)
                # Take highest confidence nodes
                sorted_nodes = sorted(nodes, key=lambda n: n.confidence, reverse=True)
                primary.extend(sorted_nodes[:self.max_citations_per_claim])
            except ValueError:
                continue

        # If no required types found, take highest confidence overall
        if not primary:
            all_nodes = list(graph._nodes.values())
            sorted_nodes = sorted(all_nodes, key=lambda n: n.confidence, reverse=True)
            primary = sorted_nodes[:5]

        return primary

    def _select_supporting_evidence(
        self,
        graph: EvidenceGraph,
        primary_nodes: list[EvidenceNode],
    ) -> list[EvidenceNode]:
        """Select supporting evidence that corroborates primary."""
        supporting = []
        primary_ids = {n.node_id for n in primary_nodes}

        for node in primary_nodes:
            support = graph.find_supporting_evidence(node.node_id, max_depth=2)
            for supp_node, depth in support:
                if supp_node.node_id not in primary_ids:
                    supporting.append(supp_node)

        # Deduplicate and limit
        seen = set()
        unique_supporting = []
        for n in supporting:
            if n.node_id not in seen:
                seen.add(n.node_id)
                unique_supporting.append(n)

        return unique_supporting[: 10]

    def _node_to_citation(self, node: EvidenceNode) -> Citation:
        """Convert evidence node to citation."""
        value_summary = self._summarize_content(node.content)

        return Citation(
            node_id=node.node_id,
            evidence_type=node.evidence_type.value,
            value_summary=value_summary,
            confidence=node.confidence,
            source_method=node.source_method,
            document_reference=node.document_location,
        )

    def _summarize_content(self, content:  dict[str, Any]) -> str:
        """Generate brief summary of evidence content."""
        # Try common fields
        for key in ["value", "text", "description", "name", "indicator"]:
            if key in content:
                val = content[key]
                if isinstance(val, str):
                    return val[: 100] + ("..." if len(val) > 100 else "")
                return str(val)[:100]

        # Fallback:  first string value
        for val in content.values():
            if isinstance(val, str) and val:
                return val[:100]

        return str(content)[:100]

    def _infer_answer_type(self, question:  str) -> str:
        """Infer answer type from question text."""
        q_lower = question.lower()

        if any(q in q_lower for q in ["Â¿cuÃ¡nto", "Â¿cuÃ¡ntos", "Â¿quÃ© porcentaje", "Â¿cuÃ¡l es el monto"]):
            return "quantitative"
        if any(q in q_lower for q in ["Â¿existe", "Â¿hay", "Â¿tiene", "Â¿incluye", "Â¿contempla"]):
            return "yes_no"
        if any(q in q_lower for q in ["Â¿cÃ³mo se compara", "Â¿cuÃ¡l es mejor", "Â¿quÃ© diferencia"]):
            return "comparative"
        return "descriptive"

    def _generate_direct_answer(
        self,
        graph: EvidenceGraph,
        question:  str,
        answer_type:  str,
        completeness: AnswerCompleteness,
        citations: list[Citation],
    ) -> str:
        """Generate the direct answer to the question."""
        n_evidence = graph.node_count
        n_citations = len(citations)

        if completeness == AnswerCompleteness.INSUFFICIENT:
            return (
                f"**No se puede responder con confianza. ** El anÃ¡lisis del documento "
                f"no produjo suficiente evidencia para responder la pregunta: "
                f"'{question[: 100]}... '.  Se identificaron solo {n_evidence} elementos "
                f"de evidencia, ninguno de los cuales cumple con los requisitos mÃnimos."
            )

        if answer_type == "yes_no":
            if n_citations > 0:
                conf_avg = statistics.mean(c.confidence for c in citations)
                if conf_avg >= 0.7:
                    return (
                        f"**SÃ**, el documento contiene evidencia positiva.  "
                        f"Se identificaron {n_citations} elementos que sustentan "
                        f"una respuesta afirmativa con confianza promedio del {conf_avg*100:.0f}%."
                    )
                else:
                    return (
                        f"**Parcialmente sÃ**, aunque con reservas.Se encontrÃ³ evidencia "
                        f"({n_citations} elementos), pero la confianza promedio es {conf_avg*100:.0f}%, "
                        f"lo que sugiere informaciÃ³n incompleta o ambigua."
                    )
            return (
                f"**No se encontrÃ³ evidencia explÃcita** que responda afirmativamente.  "
                f"El documento analizado no contiene los elementos requeridos."
            )

        elif answer_type == "quantitative":
            # Look for numeric values in citations
            numeric_vals = []
            for c in citations:
                try:
                    # Try to extract number from value_summary
                    nums = re.findall(r'[\d,. ]+', c.value_summary)
                    if nums:
                        numeric_vals.append((c.evidence_type, nums[0]))
                except (AttributeError, TypeError):
                    # If value_summary is missing or not a string, skip this citation.
                    pass

            if numeric_vals:
                primary = numeric_vals[0]
                return (
                    f"El documento reporta **{primary[1]}** para {primary[0]}. "
                    f"Esta cifra se basa en {len(numeric_vals)} indicador(es) cuantitativo(s) "
                    f"identificados en el anÃ¡lisis."
                )
            return (
                f"El documento no especifica valores numÃ©ricos precisos para esta pregunta.  "
                f"Se encontraron {n_citations} elementos de evidencia cualitativa que "
                f"pueden proporcionar contexto, pero no cifras exactas."
            )

        else:  # descriptive or comparative
            type_summary = ", ".join(set(c.evidence_type for c in citations[: 5]))

            quality = (
                "completa" if completeness == AnswerCompleteness.COMPLETE
                else "parcial"
            )

            return (
                f"El anÃ¡lisis del documento proporciona una respuesta **{quality}**. "
                f"Se identificaron {n_evidence} elementos de evidencia en categorÃas como: "
                f"{type_summary}. "
                f"{'La informaciÃ³n permite una evaluaciÃ³n confiable.' if completeness == AnswerCompleteness.COMPLETE else 'Se
 requiere informaciÃ³n adicional para una evaluaciÃ³n completa.'}"
            )

    def _build_narrative_blocks(
        self,
        direct_answer: str,
        graph: EvidenceGraph,
        completeness: AnswerCompleteness,
        validation: ValidationReport,
        primary_citations:  list[Citation],
        supporting_citations: list[Citation],
    ) -> list[NarrativeBlock]:
        """Build complete narrative structure."""
        blocks = []

        # 1. Direct Answer
        blocks.append(NarrativeBlock(
            section=NarrativeSection.DIRECT_ANSWER,
            content=direct_answer,
            citations=primary_citations[: 3],
            confidence=validation.consistency_score,
        ))

        # 2. Evidence Summary
        stats = graph.get_statistics()
        summary_content = (
            f"El anÃ¡lisis procesÃ³ evidencia de {stats['node_count']} elementos "
            f"con {stats['edge_count']} relaciones identificadas. "
            f"Confianza promedio: {stats['average_confidence']*100:.0f}%.  "
            f"DistribuciÃ³n por tipo: {self._format_type_distribution(stats['by_evidence_type'])}."
        )
        blocks.append(NarrativeBlock(
            section=NarrativeSection.EVIDENCE_SUMMARY,
            content=summary_content,
            citations=[],
            confidence=stats['average_confidence'],
        ))

        # 3. Confidence Statement
        conf_level = self._confidence_level_label(validation.consistency_score)
        conf_content = (
            f"**Nivel de confianza:  {conf_level}** ({validation.consistency_score*100:.0f}%). "
            f"Esta evaluaciÃ³n se basa en {len(primary_citations)} elementos de evidencia primaria "
            f"y {len(supporting_citations)} elementos de soporte. "
            f"{'La validaciÃ³n no reportÃ³ errores crÃticos.' if validation.is_valid else f'Se identificaron {validation.criti
cal_count} hallazgos crÃticos que afectan la confianza.'}"
        )
        blocks.append(NarrativeBlock(
            section=NarrativeSection.CONFIDENCE_STATEMENT,
            content=conf_content,
            citations=[],
            confidence=validation.consistency_score,
        ))

        # 4. Supporting Details (if sufficient evidence)
        if primary_citations:
            details_parts = []
            for citation in primary_citations[: 5]:
                rendered = citation.render("markdown")
                details_parts.append(f"- {rendered}")

            details_content = (
                "**Evidencia principal identificada:**\n" +
                "\n".join(details_parts)
            )
            blocks.append(NarrativeBlock(
                section=NarrativeSection.SUPPORTING_DETAILS,
                content=details_content,
                citations=primary_citations[: 5],
                confidence=statistics.mean(c.confidence for c in primary_citations) if primary_citations else 0.0,
            ))

        return blocks

    def _identify_gaps(
        self,
        graph: EvidenceGraph,
        expected_elements: list[dict[str, Any]],
    ) -> list[str]:
        """Identify and describe evidence gaps."""
        gaps = []

        for elem in expected_elements:
            elem_type = elem.get("type", "")
            required = elem.get("required", False)
            minimum = elem.get("minimum", 0)

            try:
                ev_type = EvidenceType(elem_type)
                nodes = graph.get_nodes_by_type(ev_type)
                count = len(nodes)

                if required and count == 0:
                    gaps.append(
                        f"**{self._humanize_type(elem_type)}** (requerido): "
                        f"No se encontrÃ³ evidencia de este tipo en el documento."
                    )
                elif minimum > 0 and count < minimum:
                    gaps.append(
                        f"**{self._humanize_type(elem_type)}**: "
                        f"Se encontraron {count} de {minimum} elementos mÃnimos requeridos."
                    )
            except ValueError:
                continue

        # Check for low-confidence evidence clusters
        low_conf_types:  dict[str, int] = defaultdict(int)
        for node in graph._nodes.values():
            if node.confidence < 0.5:
                low_conf_types[node.evidence_type.value] += 1

        for ev_type, count in low_conf_types.items():
            if count >= 3:
                gaps.append(
                    f"**{self._humanize_type(ev_type)}**: "
                    f"{count} elementos tienen confianza baja (<50%), "
                    f"lo que sugiere extracciÃ³n ambigua o fuentes poco claras."
                )

        return gaps

    def _format_contradictions(
        self,
        contradictions: list[tuple[EvidenceNode, EvidenceNode, EvidenceEdge]],
    ) -> list[str]:
        """Format contradictions for narrative."""
        formatted = []
        for source, target, edge in contradictions[: 5]:  # Limit to 5
            formatted.append(
                f"ContradicciÃ³n entre '{self._summarize_content(source.content)[: 50]}' "
                f"y '{self._summarize_content(target.content)[:50]}' "
                f"(confianza del conflicto: {edge.confidence*100:.0f}%)"
            )

        return formatted

    def _compute_confidence(
        self,
        graph:  EvidenceGraph,
        validation: ValidationReport,
        completeness: AnswerCompleteness,
    ) -> tuple[float, tuple[float, float]]:
        """
        Compute overall confidence with calibrated interval.

        Returns (point_estimate, (lower_95, upper_95))
        """
        # Base confidence from validation
        base = validation.consistency_score

        # Adjust for completeness
        completeness_factor = {
            AnswerCompleteness.COMPLETE: 1.0,
            AnswerCompleteness.PARTIAL: 0.7,
            AnswerCompleteness.INSUFFICIENT: 0.3,
            AnswerCompleteness.NOT_APPLICABLE: 0.0,
        }[completeness]

        # Adjust for evidence quantity (diminishing returns)
        quantity_factor = min(1.0, math.log1p(graph.node_count) / math.log1p(50))

        # Combine factors
        point_estimate = base * completeness_factor * (0.5 + 0.5 * quantity_factor)
        point_estimate = max(0.0, min(1.0, point_estimate))

        # Calibrated interval (Wilson score interval approximation)
        n = max(1, graph.node_count)
        z = 1.96  # 95% CI

        denominator = 1 + z**2 / n
        center = (point_estimate + z**2 / (2*n)) / denominator
        margin = z * math.sqrt((point_estimate * (1 - point_estimate) + z**2 / (4*n)) / n) / denominator

        lower = max(0.0, center - margin)
        upper = min(1.0, center + margin)

        return point_estimate, (lower, upper)

    def _format_type_distribution(self, type_counts: dict[str, int]) -> str:
        """Format type distribution for narrative."""
        if not type_counts:
            return "ninguno"

        sorted_types = sorted(type_counts.items(), key=lambda x: x[1], reverse=True)
        parts = [f"{self._humanize_type(t)}({c})" for t, c in sorted_types[: 4]]
        return ", ".join(parts)

    def _confidence_level_label(self, score: float) -> str:
        """Map confidence score to human label."""
        if score >= 0.85:
            return "ALTO"
        elif score >= 0.70:
            return "MEDIO-ALTO"
        elif score >= 0.50:
            return "MEDIO"
        elif score >= 0.30:
            return "BAJO"
        else:
            return "MUY BAJO"

    @staticmethod
    def _humanize_type(elem_type: str) -> str:
        """Convert element type to human-readable label."""
        mappings = {
            "indicador_cuantitativo": "indicadores cuantitativos",
            "serie_temporal": "series temporales",
            "monto_presupuestario": "montos presupuestarios",
            "metrica_cobertura": "mÃ©tricas de cobertura",
            "meta_cuantificada": "metas cuantificadas",
            "fuente_oficial": "fuentes oficiales",
            "cobertura_territorial": "cobertura territorial",
            "actor_institucional": "actores institucionales",
            "instrumento_politica": "instrumentos de polÃtica",
            "referencia_normativa": "referencias normativas",
            "vinculo_causal": "vÃnculos causales",
            "dependencia_temporal": "dependencias temporales",
            "contradiccion": "contradicciones",
            "corroboracion": "corroboraciones",
            "fuentes_oficiales": "fuentes oficiales",
            "indicadores_cuantitativos": "indicadores cuantitativos",
            "series_temporales_aÃ±os": "series temporales",
            "cobertura_territorial_especificada": "cobertura territorial",
        }
        return mappings.get(elem_type, elem_type.replace("_", " "))


# ============================================================================
# UNIFIED ENGINE:  EvidenceNexus
# ============================================================================

class EvidenceNexus:
    """
    Unified SOTA Evidence-to-Answer Engine.

    REPLACES:
    - EvidenceAssembler:  Graph-based evidence fusion
    - EvidenceValidator: Probabilistic graph validation
    - EvidenceRegistry: Embedded provenance with hash chain

    PROVIDES:
    - Causal graph construction from method outputs
    - Bayesian belief propagation
    - Conflict detection and resolution
    - Narrative synthesis with citations
    - Cryptographic provenance chain

    Usage:
        nexus = EvidenceNexus()
        result = nexus.process(
            method_outputs=method_outputs,
            question_context=question_context,
            contract=contract,
        )

        # Result contains:
        # - evidence_graph: Full graph with all nodes/edges
        # - validation_report: Comprehensive validation
        # - synthesized_answer: Complete narrative answer
        # - human_readable_output: Formatted string
    """

    def __init__(
        self,
        storage_path: Path | None = None,
        enable_persistence: bool = True,
        validation_rules: list[ValidationRule] | None = None,
        citation_threshold: float = 0.6,
    ):
        """
        Initialize EvidenceNexus.

        Args:
            storage_path: Path for persistent storage (JSONL)
            enable_persistence: Whether to persist to disk
            validation_rules:  Custom validation rules
            citation_threshold:  Minimum confidence for citation
        """
        self.storage_path = storage_path or Path("evidence_nexus.jsonl")
        self.enable_persistence = enable_persistence

        self.validation_engine = ValidationEngine(rules=validation_rules)
        self.narrative_synthesizer = NarrativeSynthesizer(
            citation_threshold=citation_threshold
        )

        # Current session graph
        self._graph:  EvidenceGraph | None = None

        logger.info(
            "evidence_nexus_initialized",
            storage_path=str(self.storage_path),
            persistence=enable_persistence,
        )

    def process(
        self,
        method_outputs: dict[str, Any],
        question_context: dict[str, Any],
        contract: dict[str, Any],
        signal_pack: Any | None = None,
    ) -> dict[str, Any]:
        """
        Process method outputs into complete answer.

        This is the main entry point that replaces:
        - EvidenceAssembler.assemble()
        - EvidenceValidator.validate()
        - EvidenceRegistry.record_evidence()

        Args:
            method_outputs: Raw outputs from executor methods
            question_context: Question context with expected_elements
            contract: Full v3 contract
            signal_pack: Optional signal pack for provenance

        Returns:
            Complete result dict with:
            - evidence:  Assembled evidence (legacy compatible)
            - validation: Validation results (legacy compatible)
            - trace: Execution trace (legacy compatible)
            - synthesized_answer: New narrative answer
            - human_readable_output: Formatted answer string
            - graph_statistics: Graph metrics
        """
        start_time = time.time()

        # 1. Build evidence graph from method outputs
        graph = self._build_graph_from_outputs(
            method_outputs, question_context, contract, signal_pack
        )
        self._graph = graph

        # 2. Infer relationships between evidence nodes
        self._infer_relationships(graph, contract)

        # 3. Run belief propagation
        beliefs = graph.compute_belief_propagation()

        # 4. Validate graph
        validation_report = self.validation_engine.validate(graph, contract)

        # 5. Synthesize narrative answer
        synthesized = self.narrative_synthesizer.synthesize(
            graph, question_context, validation_report, contract
        )

        # 6. Persist if enabled
        if self.enable_persistence:
            self._persist_graph(graph)

        # 7. Build legacy-compatible evidence dict
        legacy_evidence = self._build_legacy_evidence(graph, beliefs)
        legacy_validation = self._build_legacy_validation(validation_report)
        legacy_trace = self._build_legacy_trace(graph, signal_pack)

        # -----------------------------------------------------------------
        # SISAS utility / consumption proof (if injected by executor)
        # -----------------------------------------------------------------
        tracker = question_context.get("consumption_tracker")
        if tracker is not None:
            try:
                # ConsumptionTracker provides a summary and an embedded proof object
                if hasattr(tracker, "get_consumption_summary"):
                    legacy_trace["signal_consumption"] = tracker.get_consumption_summary()
                if hasattr(tracker, "get_proof"):
                    proof = tracker.get_proof()
                    if hasattr(proof, "get_consumption_proof"):
                        legacy_trace["signal_consumption_proof"] = proof.get_consumption_proof()
            except Exception:
                # Never break processing for telemetry failures
                pass

        processing_time_ms = (time.time() - start_time) * 1000

        logger.info(
            "evidence_nexus_process_complete",
            node_count=graph.node_count,
            edge_count=graph.edge_count,
            is_valid=validation_report.is_valid,
            completeness=synthesized.completeness.value,
            confidence=f"{synthesized.overall_confidence:.2f}",
            processing_time_ms=f"{processing_time_ms:.1f}",
        )

        return {
            # Legacy compatible
            "evidence": legacy_evidence,
            "validation": legacy_validation,
            "trace": legacy_trace,

            # New SOTA outputs
            "synthesized_answer":  synthesized.to_dict(),
            "human_readable_output": synthesized.to_human_readable("markdown"),
            "direct_answer": synthesized.direct_answer,

            # Graph data
            "graph_statistics": graph.get_statistics(),
            "graph_hash": graph.get_graph_hash(),

            # Metrics
            "completeness": synthesized.completeness.value,
            "overall_confidence": synthesized.overall_confidence,
            "calibrated_interval": list(synthesized.calibrated_interval),
            "gaps": synthesized.gaps,
            "contradictions": synthesized.unresolved_contradictions,

            # Processing metadata
            "processing_time_ms": processing_time_ms,
            "nexus_version": "1.0.0",
        }

    def _build_graph_from_outputs(
        self,
        method_outputs: dict[str, Any],
        question_context: dict[str, Any],
        contract: dict[str, Any],
        signal_pack: Any | None,
    ) -> EvidenceGraph:
        """
        Transform method outputs into evidence graph.

        Replaces EvidenceAssembler’s merge logic with graph construction.
        """
        graph = EvidenceGraph()

        # ---------------------------------------------------------------------
        # Pattern-derived evidence (contract patterns)
        # ---------------------------------------------------------------------
        # v3 executors pass patterns + raw_text into question_context so that
        # patterns add value even when downstream methods don’t consume them.
        raw_text = (
            question_context.get("raw_text")
            or question_context.get("text")
            or question_context.get("document_text")
        )
        patterns = (
            question_context.get("patterns")
            or contract.get("question_context", {}).get("patterns", [])
        )
        if isinstance(raw_text, str) and raw_text and isinstance(patterns, list) and patterns:
            graph.add_nodes(
                self._extract_nodes_from_contract_patterns(
                    raw_text=raw_text,
                    patterns=patterns,
                    question_context=question_context,
                )
            )

        # Get assembly rules from contract
        evidence_assembly = contract.get("evidence_assembly", {})
        assembly_rules = evidence_assembly.get("assembly_rules", [])

        # Process each method output
        for source_key, output in method_outputs.items():
            if source_key.startswith("_"):
                continue  # Skip internal keys like _signal_usage

            nodes = self._extract_nodes_from_output(
                source_key, output, question_context
            )
            graph.add_nodes(nodes)

        # Apply assembly rules to create aggregate nodes
        for rule in assembly_rules:
            target = rule.get("target")
            sources = rule.get("sources", [])
            strategy = rule.get("merge_strategy", "concat")

            aggregate_node = self._create_aggregate_node(
                target, sources, strategy, graph, method_outputs
            )
            if aggregate_node:
                graph.add_node(aggregate_node)

        # Add signal provenance if available
        if signal_pack is not None:
            provenance_node = self._create_provenance_node(signal_pack)
            graph.add_node(provenance_node)

        return graph

    def _extract_nodes_from_contract_patterns(
        self,
        *,
        raw_text: str,
        patterns: list[Any],
        question_context: dict[str, Any],
        max_matches_per_pattern: int = 5,
    ) -> list[EvidenceNode]:
        """Create evidence nodes from v3 contract patterns.

        Goal: patterns contribute to evidence/scoring even if methods ignore them.

        This is intentionally conservative:
        - Only regex/literal matching (NER_OR_REGEX treated as regex fallback)
        - Caps matches per pattern for determinism and bounded output
        """
        nodes: list[EvidenceNode] = []
        qid = str(question_context.get("question_id") or "")
        document_context = question_context.get("document_context")
        if not isinstance(document_context, dict):
            document_context = {}

        # Optional SISAS consumption tracker (utility measurement + proof chain)
        tracker = question_context.get("consumption_tracker")

        # Optional context-aware filtering from SISAS
        filtered_patterns = patterns
        context_filter_stats: dict[str, int] | None = None
        if document_context:
            try:
                from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_context_scoper import (
                    filter_patterns_by_context,
                )

                # filter_patterns_by_context expects list[dict], ignore non-dicts
                dict_patterns = [p for p in patterns if isinstance(p, dict)]
                filtered_patterns, context_filter_stats = filter_patterns_by_context(
                    dict_patterns, document_context
                )
            except Exception:
                filtered_patterns = patterns
                context_filter_stats = None

        # Optional semantic expansion from SISAS (Refactoring #2)
        expanded_patterns = filtered_patterns
        expansion_stats: dict[str, Any] | None = None
        try:
            from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_semantic_expander import (
                expand_all_patterns,
                validate_expansion_result,
            )

            dict_patterns = [p for p in filtered_patterns if isinstance(p, dict)]
            if dict_patterns:
                expanded_patterns = expand_all_patterns(dict_patterns, enable_logging=False)
                try:
                    expansion_stats = validate_expansion_result(
                        dict_patterns,
                        expanded_patterns,
                        min_multiplier=2.0,
                        target_multiplier=5.0,
                    )
                except Exception:
                    expansion_stats = None
        except Exception:
            expanded_patterns = filtered_patterns
            expansion_stats = None

        max_patterns = 2000
        if isinstance(expanded_patterns, list) and len(expanded_patterns) > max_patterns:
            expanded_patterns = expanded_patterns[:max_patterns]

        def _to_flags(flags_value: Any) -> int:
            if not flags_value:
                return re.IGNORECASE | re.MULTILINE
            if isinstance(flags_value, int):
                return flags_value
            if isinstance(flags_value, str):
                f = 0
                if "i" in flags_value:
                    f |= re.IGNORECASE
                if "m" in flags_value:
                    f |= re.MULTILINE
                if "s" in flags_value:
                    f |= re.DOTALL
                return f or (re.IGNORECASE | re.MULTILINE)
            return re.IGNORECASE | re.MULTILINE

        def _map_category_to_evidence_type(category: str) -> EvidenceType:
            cat = (category or "").upper()
            if cat == "INDICADOR" or cat == "UNIDAD_MEDIDA":
                return EvidenceType.INDICATOR_NUMERIC
            if cat == "TEMPORAL":
                return EvidenceType.TEMPORAL_SERIES
            if cat == "FUENTE_OFICIAL":
                return EvidenceType.OFFICIAL_SOURCE
            if cat == "TERRITORIAL":
                return EvidenceType.TERRITORIAL_COVERAGE
            if cat.startswith("CAUSAL"):
                return EvidenceType.CAUSAL_LINK
            # Default: treat as method-output-like evidence with tags
            return EvidenceType.METHOD_OUTPUT

        for pat in expanded_patterns:
            if not isinstance(pat, dict):
                continue

            pattern_id = str(pat.get("id") or pat.get("pattern_id") or pat.get("pattern_ref") or "")
            category = str(pat.get("category") or "GENERAL")
            match_type = str(pat.get("match_type") or "REGEX").upper()
            pattern_str = pat.get("pattern")

            # If pattern_ref exists but pattern is missing, we still emit a trace node
            # so the deficiency is visible in downstream telemetry.
            if not isinstance(pattern_str, str) or not pattern_str.strip():
                node = EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
                    content={
                        "pattern_id": pattern_id,
                        "pattern_ref": pat.get("pattern_ref"),
                        "category": category,
                        "match_type": match_type,
                        "matches": [],
                        "note": "pattern_missing_in_contract",
                        "question_id": qid,
                    },
                    confidence=0.1,
                    source_method="contract.patterns",
                    tags=["contract_pattern", "pattern_missing"],
                )
                nodes.append(node)
                continue

            flags_int = _to_flags(pat.get("flags"))
            matches: list[str] = []
            try:
                if match_type == "LITERAL":
                    # Simple containment; return matched literal as-is once per occurrence (bounded).
                    literal = pattern_str
                    start = 0
                    while len(matches) < max_matches_per_pattern:
                        idx = raw_text.lower().find(literal.lower(), start)
                        if idx < 0:
                            break
                        matches.append(raw_text[idx : idx + len(literal)])
                        start = idx + len(literal)
                else:
                    compiled = re.compile(pattern_str, flags_int)
                    for m in compiled.finditer(raw_text):
                        matches.append(m.group(0))
                        if len(matches) >= max_matches_per_pattern:
                            break
            except Exception:
                # Invalid regex patterns should not crash pipeline; emit diagnostic node.
                node = EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
                    content={
                        "pattern_id": pattern_id,
                        "pattern_ref": pat.get("pattern_ref"),
                        "category": category,
                        "match_type": match_type,
                        "pattern": pattern_str,
                        "matches": [],
                        "note": "pattern_compile_or_match_failed",
                        "question_id": qid,
                    },
                    confidence=0.1,
                    source_method="contract.patterns",
                    tags=["contract_pattern", "pattern_error"],
                )
                nodes.append(node)
                continue

            # Record consumption proof (if tracker injected)
            if tracker is not None and matches:
                try:
                    # Track up to the same cap; produced_evidence=True because we are emitting nodes
                    for mtxt in matches:
                        # ConsumptionTracker API: record_pattern_match(pattern, text_segment, produced_evidence)
                        if hasattr(tracker, "record_pattern_match"):
                            tracker.record_pattern_match(
                                pattern={"id": pattern_id, "pattern": pattern_str},
                                text_segment=str(mtxt),
                                produced_evidence=True,
                            )
                except Exception:
                    # Never break execution for tracking failures
                    pass

            confidence_weight = pat.get("confidence_weight")
            try:
                confidence = float(confidence_weight) if confidence_weight is not None else 0.6
            except Exception:
                confidence = 0.6
            if not matches:
                confidence = min(confidence, 0.35)

            node = EvidenceNode.create(
                evidence_type=_map_category_to_evidence_type(category),
                content={
                    "pattern_id": pattern_id,
                    "pattern_ref": pat.get("pattern_ref"),
                    "category": category,
                    "match_type": match_type,
                    "pattern": pattern_str,
                    "matches": matches,
                    "match_count": len(matches),
                    "question_id": qid,
                },
                confidence=max(0.0, min(1.0, confidence)),
                source_method="contract.patterns",
                tags=["contract_pattern", category.lower()],
            )
            nodes.append(node)

        # Emit a lightweight stats node for context filtering and utility accounting
        try:
            total_injected = len([p for p in patterns if isinstance(p, dict)])
            total_after_context = len([p for p in filtered_patterns if isinstance(p, dict)])
            total_considered = len([p for p in expanded_patterns if isinstance(p, dict)])
            matched_patterns = 0
            for n in nodes:
                if isinstance(n.content, dict) and int(n.content.get("match_count", 0) or 0) > 0:
                    matched_patterns += 1
            waste_ratio = (
                1.0 - (matched_patterns / total_considered)
                if total_considered > 0
                else 1.0
            )
            nodes.append(
                EvidenceNode.create(
                    evidence_type=EvidenceType.METHOD_OUTPUT,
                    content={
                        "provenance_type": "contract_pattern_utility",
                        "question_id": qid,
                        "patterns_injected": total_injected,
                        "patterns_after_context_filter": total_after_context,
                        "patterns_considered": total_considered,
                        "patterns_matched": matched_patterns,
                        "waste_ratio": round(float(waste_ratio), 4),
                        "context_filter_stats": context_filter_stats,
                        "semantic_expansion_stats": expansion_stats,
                    },
                    confidence=1.0,
                    source_method="contract.patterns.utility",
                    tags=["contract_pattern", "utility"],
                )
            )
        except Exception:
            pass

        return nodes

    def _extract_nodes_from_output(
        self,
        source_key: str,
        output: Any,
        question_context: dict[str, Any],
    ) -> list[EvidenceNode]:
        """Extract evidence nodes from a single method output."""
        nodes = []

        if output is None:
            return nodes

        # Handle list outputs (multiple evidence items)
        if isinstance(output, list):
            for idx, item in enumerate(output):
                node = self._item_to_node(
                    item,
                    source_method=source_key,
                    index=idx,
                )
                if node:
                    nodes.append(node)

        # Handle dict outputs (structured evidence)
        elif isinstance(output, dict):
            # Check if it’s a single evidence item or container
            if "elements" in output:
                # Container with elements list
                for idx, item in enumerate(output.get("elements", [])):
                    node = self._item_to_node(
                        item,
                        source_method=source_key,
                        index=idx,
                    )
                    if node:
                        nodes.append(node)
            else:
                # Single evidence item
                node = self._item_to_node(
                    output,
                    source_method=source_key,
                )
                if node:
                    nodes.append(node)

        # Handle scalar outputs
        else:
            node = EvidenceNode.create(
                evidence_type=EvidenceType.METHOD_OUTPUT,
                content={"value": output, "source":  source_key},
                confidence=0.7,  # Default for raw method output
                source_method=source_key,
            )
            nodes.append(node)

        return nodes

    def _item_to_node(
        self,
        item: Any,
        source_method: str,
        index: int | None = None,
    ) -> EvidenceNode | None:
        """Convert a single item to evidence node."""
        if item is None:
            return None

        if isinstance(item, dict):
            # Extract type
            item_type = item.get("type", item.get("evidence_type", ""))
            try:
                ev_type = EvidenceType(item_type)
            except ValueError:
                ev_type = EvidenceType.METHOD_OUTPUT

            # Extract confidence
            confidence = item.get("confidence", item.get("score", 0.7))
            if isinstance(confidence, str):
                try:
                    confidence = float(confidence.strip("%")) / 100
                except (ValueError, TypeError):
                    confidence = 0.7

            # Extract document location
            doc_loc = item.get("page", item.get("location", item.get("section")))
            if doc_loc is not None:
                doc_loc = str(doc_loc)

            return EvidenceNode.create(
                evidence_type=ev_type,
                content=item,
                confidence=float(confidence),
                source_method=source_method,
                document_location=doc_loc,
                tags=frozenset(item.get("tags", [])),
            )

        # Non-dict item
        return EvidenceNode.create(
            evidence_type=EvidenceType.METHOD_OUTPUT,
            content={"value": item, "index": index},
            confidence=0.6,
            source_method=source_method,
        )

    def _create_aggregate_node(
        self,
        target: str,
        sources:  list[str],
        strategy:  str,
        graph: EvidenceGraph,
        method_outputs: dict[str, Any],
    ) -> EvidenceNode | None:
        """Create aggregate node from assembly rule."""
        # Collect source values
        values = []
        parent_ids = []

        for source in sources:
            # Resolve dotted path
            value = self._resolve_path(source, method_outputs)
            if value is not None:
                values.append(value)
                # Find corresponding nodes
                for node in graph._nodes.values():
                    # Source keys in method_outputs are dotted paths (no space after ’.’)
                    if node.source_method == source.split(".")[0]:
                        parent_ids.append(node.node_id)

        if not values:
            return None

        # Apply merge strategy
        merged_value = self._apply_merge_strategy(values, strategy)

        return EvidenceNode.create(
            evidence_type=EvidenceType.AGGREGATED,
            content={
                "target": target,
                "strategy": strategy,
                "sources":  sources,
                "value": merged_value,
            },
            confidence=0.8,  # Aggregated confidence
            source_method=f"aggregate:{target}",
            parent_ids=parent_ids[: 10],  # Limit parents
        )

    def _resolve_path(self, path: str, data: dict[str, Any]) -> Any:
        """Resolve dotted path in data structure."""
        parts = path.split(".")
        current = data

        for part in parts:
            if isinstance(current, dict) and part in current:
                current = current[part]
            else:
                return None

        return current
```

Phase 2 Source Code

```
    def _apply_merge_strategy(
        self,
        values: list[Any],
        strategy: str,
    ) -> Any:
        """Apply merge strategy to values."""
        if not values:
            return None

        if strategy == "first":
            return values[0]
        elif strategy == "last":
            return values[-1]
        elif strategy == "concat":
            result = []
            for v in values:
                if isinstance(v, list):
                    result.extend(v)
                else:
                    result.append(v)
            return result
        elif strategy == "mean":
            numeric = [float(v) for v in values if self._is_numeric(v)]
            return statistics.mean(numeric) if numeric else None
        elif strategy == "max":
            numeric = [float(v) for v in values if self._is_numeric(v)]
            return max(numeric) if numeric else None
        elif strategy == "min":
            numeric = [float(v) for v in values if self._is_numeric(v)]
            return min(numeric) if numeric else None
        elif strategy == "weighted_mean":
            numeric = [float(v) for v in values if self._is_numeric(v)]
            return statistics.mean(numeric) if numeric else None
        elif strategy == "majority":
            from collections import Counter
            counts = Counter(str(v) for v in values)
            return counts.most_common(1)[0][0] if counts else None
        else:
            return values[0]  # Default to first

    @staticmethod
    def _is_numeric(value: Any) -> bool:
        """Check if value is numeric."""
        if isinstance(value, bool):
            return False
        try:
            float(value)
            return True
        except (TypeError, ValueError):
            return False

    def _create_provenance_node(self, signal_pack: Any) -> EvidenceNode:
        """Create provenance node from signal pack."""
        pack_id = getattr(signal_pack, "id", None) or getattr(signal_pack, "pack_id", "unknown")
        policy_area = getattr(signal_pack, "policy_area", None)
        version = getattr(signal_pack, "version", "unknown")

        return EvidenceNode.create(
            evidence_type=EvidenceType.METHOD_OUTPUT,
            content={
                "provenance_type": "signal_pack",
                "pack_id": pack_id,
                "policy_area": str(policy_area) if policy_area else None,
                "version": version,
            },
            confidence=1.0,  # Provenance is certain
            source_method="signal_registry",
            tags=frozenset(["provenance", "signal_pack"]),
        )

    def _infer_relationships(
        self,
        graph: EvidenceGraph,
        contract:  dict[str, Any],
    ) -> None:
        """
        Infer relationships between evidence nodes.

        Uses:
        - Type compatibility for SUPPORTS edges
        - Temporal ordering for PRECEDES edges
        - Content similarity for CORRELATES edges
        - Contradiction detection for CONTRADICTS edges
        """
        nodes = list(graph._nodes.values())

        # Infer DERIVES_FROM from parent_ids
        for node in nodes:
            for parent_id in node.parent_ids:
                if parent_id in graph._nodes:
                    edge = EvidenceEdge.create(
                        source_id=parent_id,
                        target_id=node.node_id,
                        relation_type=RelationType.DERIVES_FROM,
                        weight=0.9,
                        confidence=0.95,
                )
                try:
                    graph.add_edge(edge)
                except ValueError:
                    pass  # Skip if would create cycle

        # Infer SUPPORTS between related types
        support_pairs = [
            (EvidenceType.OFFICIAL_SOURCE, EvidenceType.INDICATOR_NUMERIC),
            (EvidenceType.INDICATOR_NUMERIC, EvidenceType.GOAL_TARGET),
            (EvidenceType.BUDGET_AMOUNT, EvidenceType.POLICY_INSTRUMENT),
        ]

        for source_type, target_type in support_pairs:
            source_nodes = graph.get_nodes_by_type(source_type)
            target_nodes = graph.get_nodes_by_type(target_type)

            for sn in source_nodes[: 5]:  # Limit to prevent explosion
                for tn in target_nodes[:5]:
                    if sn.node_id != tn.node_id:
                        edge = EvidenceEdge.create(
                            source_id=sn.node_id,
                            target_id=tn.node_id,
                            relation_type=RelationType.SUPPORTS,
                            weight=0.6,
                            confidence=0.7,
                        )
                        try:
                            graph.add_edge(edge)
                        except ValueError:
                            pass  # Skip if adding SUPPORTS edge would create cycle or is invalid

    def _persist_graph(self, graph: EvidenceGraph) -> None:
        """Persist graph to storage."""
        if not self.enable_persistence:
            return

        try:
            self.storage_path.parent.mkdir(parents=True, exist_ok=True)

            with open(self.storage_path, "a", encoding="utf-8") as f:
                # Write summary record
                record = {
                    "timestamp":  time.time(),
                    "graph_hash": graph.get_graph_hash(),
                    "node_count": graph.node_count,
                    "edge_count": graph.edge_count,
                    "statistics": graph.get_statistics(),
                }
                f.write(json.dumps(record, separators=(",", ":")) + "\n")

            logger.debug("graph_persisted", path=str(self.storage_path))

        except Exception as e:
            logger.error("graph_persistence_failed", error=str(e))

    def _build_legacy_evidence(
        self,
        graph: EvidenceGraph,
        beliefs: dict[EvidenceID, float],
    ) -> dict[str, Any]:
        """Build legacy-compatible evidence dict."""
        elements = []
        by_type:  dict[str, list[dict]] = defaultdict(list)
        confidences = []

        for node in graph._nodes.values():
            elem = {
                "element_id": node.node_id[: 12],
                "type": node.evidence_type.value,
                "value": self._extract_value(node.content),
                "confidence": node.confidence,
                "belief":  beliefs.get(node.node_id, node.confidence),
                "source_method": node.source_method,
            }
            elements.append(elem)
            by_type[node.evidence_type.value].append(elem)
            confidences.append(node.confidence)

        return {
            "elements": elements,
            "elements_found_count": len(elements),
            "by_type": {k: len(v) for k, v in by_type.items()},
            "confidence_scores": {
                "mean": statistics.mean(confidences) if confidences else 0.0,
                "min": min(confidences) if confidences else 0.0,
                "max": max(confidences) if confidences else 0.0,
            },
            "graph_hash": graph.get_graph_hash()[: 16],
        }

    def _extract_value(self, content: dict[str, Any]) -> Any:
        """Extract primary value from content dict."""
        for key in ["value", "text", "description", "name", "indicator"]:
            if key in content:
                return content[key]
        return content

    def _build_legacy_validation(
        self,
        report: ValidationReport,
    ) -> dict[str, Any]:
        """Build legacy-compatible validation dict."""
        return {
            "valid":  report.is_valid,
            "passed":  report.is_valid,
            "errors": [f.to_dict() for f in report.findings if f.severity in (ValidationSeverity.CRITICAL, ValidationSeverity.E
RROR)],
            "warnings": [f.to_dict() for f in report.findings if f.severity == ValidationSeverity.WARNING],
            "critical_count": report.critical_count,
            "error_count": report.error_count,
            "warning_count": report.warning_count,
            "consistency_score": report.consistency_score,
            "graph_integrity": report.graph_integrity,
        }

    def _build_legacy_trace(
        self,
        graph: EvidenceGraph,
        signal_pack: Any | None,
    ) -> dict[str, Any]:
        """Build legacy-compatible trace dict."""
        trace = {
            "graph_statistics": graph.get_statistics(),
            "hash_chain_length": len(graph._hash_chain),
            "processing_timestamp": time.time(),
        }

        # Contract pattern utility summary (always available if pattern extraction ran)
        try:
            utility_nodes = graph.get_nodes_by_source("contract.patterns.utility")
            if utility_nodes:
                # Keep last node (single) as authoritative
                node = utility_nodes[-1]
                if isinstance(node.content, dict):
                    trace["pattern_utility"] = {
                        "patterns_injected": node.content.get("patterns_injected"),
                        "patterns_considered": node.content.get("patterns_considered"),
                        "patterns_matched": node.content.get("patterns_matched"),
                        "waste_ratio": node.content.get("waste_ratio"),
                        "context_filter_stats": node.content.get("context_filter_stats"),
                    }
        except Exception:
            pass

        if signal_pack is not None:
            trace["signal_provenance"] = {
                "signal_pack_id": getattr(signal_pack, "id", None) or getattr(signal_pack, "pack_id", "unknown"),
                "policy_area":  str(getattr(signal_pack, "policy_area", None)),
                "version": getattr(signal_pack, "version", "unknown"),
            }

        return trace

    # -------------------------------------------------------------------------
    # Public Query Interface
    # -------------------------------------------------------------------------

    def get_current_graph(self) -> EvidenceGraph | None:
        """Get current session graph."""
        return self._graph

    def query_by_type(self, evidence_type: EvidenceType) -> list[EvidenceNode]:
        """Query nodes by type from current graph."""
        if self._graph is None:
            return []
        return self._graph.get_nodes_by_type(evidence_type)

    def query_by_source(self, source_method: str) -> list[EvidenceNode]:
        """Query nodes by source method from current graph."""
        if self._graph is None:
            return []
        return self._graph.get_nodes_by_source(source_method)

    def get_statistics(self) -> dict[str, Any]:
        """Get current graph statistics."""
        if self._graph is None:
            return {"error": "No graph in current session"}
        return self._graph.get_statistics()


# =============================================================================
# FACTORY AND CONVENIENCE FUNCTIONS
# =============================================================================

# Global instance (singleton pattern for registry compatibility)
_global_nexus: EvidenceNexus | None = None


def get_global_nexus() -> EvidenceNexus:
    """Get or create global EvidenceNexus instance."""
    global _global_nexus
    if _global_nexus is None:
        _global_nexus = EvidenceNexus()
    return _global_nexus


def process_evidence(
    method_outputs: dict[str, Any],
    question_context: dict[str, Any],
    contract: dict[str, Any],
    signal_pack: Any | None = None,
) -> dict[str, Any]:
    """
    Convenience function for one-shot evidence processing.

    This replaces the typical pattern of:
        assembled = EvidenceAssembler.assemble(...)
        validation = EvidenceValidator.validate(...)
        registry.record_evidence(...)

    With:
        result = process_evidence(...)
    """
    nexus = get_global_nexus()
    return nexus.process(
        method_outputs=method_outputs,
        question_context=question_context,
        contract=contract,
        signal_pack=signal_pack,
    )


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

__all__ = [
    # Core types
    "EvidenceType",
    "RelationType",
    "ValidationSeverity",
    "AnswerCompleteness",
    "NarrativeSection",

    # Data structures
    "EvidenceNode",
    "EvidenceEdge",
    "ValidationFinding",
    "ValidationReport",
    "Citation",
    "NarrativeBlock",
    "SynthesizedAnswer",

    # Graph
    "EvidenceGraph",

    # Engines
    "ValidationEngine",
    "NarrativeSynthesizer",

    # Main class
    "EvidenceNexus",

    # Factory functions
    "get_global_nexus",
    "process_evidence",
]



################################################################################
# FILE: phase2_g_method_signature_validator.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_g_method_signature_validator.py
################################################################################

"""
Module: phase2_g_method_signature_validator
PHASE_LABEL: Phase 2
Sequence: G

"""
"""
Method Signature Chain Layer Validation

Implements chain layer validation for method signatures to ensure:
- Required inputs are properly declared (hard failure if missing)
- Optional inputs are classified (nice to have)
- Critical optional inputs are identified (penalize if missing)
- Output types and ranges are properly specified
- Signature completeness across all methods

This module provides signature governance for the analysis pipeline.
"""

import json
from datetime import datetime
from pathlib import Path
```

Phase 2 Source Code

```
from typing import Any, TypedDict


class MethodSignature(TypedDict):
    required_inputs: list[str]
    optional_inputs: list[str]
    critical_optional: list[str]
    output_type: str
    output_range: list[float] | None
    description: str


class SignatureValidationResult(TypedDict):
    is_valid: bool
    missing_fields: list[str]
    issues: list[str]
    warnings: list[str]


class ValidationReport(TypedDict):
    validation_timestamp: str
    signatures_version: str
    total_methods: int
    valid_methods: int
    invalid_methods: int
    incomplete_methods: int
    methods_with_warnings: int
    validation_details: dict[str, SignatureValidationResult]
    summary: dict[str, Any]


class MethodSignatureValidator:
    """
    Validates method signatures for chain layer compliance.

    Ensures all methods have proper signature declarations with:
    - Required fields: required_inputs, output_type
    - Recommended fields: optional_inputs, critical_optional, output_range
    """

    REQUIRED_SIGNATURE_FIELDS = {"required_inputs", "output_type"}
    RECOMMENDED_SIGNATURE_FIELDS = {
        "optional_inputs",
        "critical_optional",
        "output_range",
    }
    ALL_SIGNATURE_FIELDS = (
        REQUIRED_SIGNATURE_FIELDS | RECOMMENDED_SIGNATURE_FIELDS | {"description"}
    )

    VALID_OUTPUT_TYPES = {"float", "int", "dict", "list", "str", "bool", "tuple", "Any"}

    def __init__(self, signatures_path: Path | str) -> None:
        self.signatures_path = Path(signatures_path)
        self.signatures_data: dict[str, Any] = {}
        self.validation_cache: dict[str, SignatureValidationResult] = {}

    def load_signatures(self) -> None:
        """Load method signatures from JSON file."""
        if not self.signatures_path.exists():
            raise FileNotFoundError(
                f"Signatures file not found: {self.signatures_path}"
            )

        with open(self.signatures_path) as f:
            self.signatures_data = json.load(f)

        if "methods" not in self.signatures_data:
            raise ValueError("Invalid signatures file: missing ’methods’ key")

    def validate_signature(
        self, method_id: str, signature: dict[str, Any]
    ) -> SignatureValidationResult:
        """
        Validate a single method signature.

        Classification:
        - required_inputs: MUST be present, hard failure if missing at runtime
        - optional_inputs: Nice to have, no penalty if missing
        - critical_optional: Penalize if missing, but don’t fail hard
        """
        is_valid = True
        missing_fields = []
        issues = []
        warnings = []

        # Check required fields
        for field in self.REQUIRED_SIGNATURE_FIELDS:
            if field not in signature:
                is_valid = False
                missing_fields.append(field)
                issues.append(f"Missing required field: {field}")

        # Check recommended fields
        for field in self.RECOMMENDED_SIGNATURE_FIELDS:
            if field not in signature:
                warnings.append(f"Missing recommended field: {field}")

        # Validate required_inputs
        if "required_inputs" in signature:
            if not isinstance(signature["required_inputs"], list):
                is_valid = False
                issues.append("required_inputs must be a list")
            elif len(signature["required_inputs"]) == 0:
                warnings.append(
                    "required_inputs is empty - method has no mandatory inputs"
                )
            else:
                # Validate input names
                for inp in signature["required_inputs"]:
                    if not isinstance(inp, str):
                        is_valid = False
                        issues.append(f"Invalid required input (not a string): {inp}")

        # Validate optional_inputs
        if "optional_inputs" in signature:
            if not isinstance(signature["optional_inputs"], list):
                is_valid = False
                issues.append("optional_inputs must be a list")
            else:
                for inp in signature["optional_inputs"]:
                    if not isinstance(inp, str):
                        is_valid = False
                        issues.append(f"Invalid optional input (not a string): {inp}")

        # Validate critical_optional
        if "critical_optional" in signature:
            if not isinstance(signature["critical_optional"], list):
                is_valid = False
                issues.append("critical_optional must be a list")
            else:
                # Check that critical_optional items are in optional_inputs
                optional_inputs = signature.get("optional_inputs", [])
                for inp in signature["critical_optional"]:
                    if not isinstance(inp, str):
                        is_valid = False
                        issues.append(
                            f"Invalid critical_optional input (not a string): {inp}"
                        )
                    elif inp not in optional_inputs:
                        warnings.append(
                            f"critical_optional input '{inp}' not found in optional_inputs"
                        )

        # Validate output_type
        if "output_type" in signature:
            output_type = signature["output_type"]
            if not isinstance(output_type, str):
                is_valid = False
                issues.append("output_type must be a string")
            elif output_type not in self.VALID_OUTPUT_TYPES:
                warnings.append(
                    f"output_type '{output_type}' not in standard types: {self.VALID_OUTPUT_TYPES}"
                )

        # Validate output_range
        if "output_range" in signature:
            output_range = signature["output_range"]
            if output_range is not None:
                if not isinstance(output_range, list):
                    is_valid = False
                    issues.append("output_range must be a list or null")
                elif len(output_range) != 2:
                    is_valid = False
                    issues.append(
                        "output_range must have exactly 2 elements [min, max]"
                    )
                else:
                    try:
                        min_val, max_val = float(output_range[0]), float(
                            output_range[1]
                        )
                        if min_val >= max_val:
                            is_valid = False
                            issues.append("output_range min must be less than max")
                    except (ValueError, TypeError):
                        is_valid = False
                        issues.append("output_range values must be numeric")

        # Check for unknown fields
        unknown_fields = set(signature.keys()) - self.ALL_SIGNATURE_FIELDS
        if unknown_fields:
            warnings.append(f"Unknown fields in signature: {unknown_fields}")

        return SignatureValidationResult(
            is_valid=is_valid,
            missing_fields=missing_fields,
            issues=issues,
            warnings=warnings,
        )

    def validate_all_signatures(self) -> ValidationReport:
```

Phase 2 Source Code

```
        """Validate all method signatures and generate comprehensive report."""
        if not self.signatures_data:
            self.load_signatures()

        methods = self.signatures_data.get("methods", {})
        validation_details: dict[str, SignatureValidationResult] = {}

        valid_count = 0
        invalid_count = 0
        incomplete_count = 0
        warnings_count = 0

        for method_id, method_data in methods.items():
            # Handle both flat structure and nested signature structure
            if "signature" in method_data:
                signature = method_data["signature"]
            else:
                signature = method_data

            result = self.validate_signature(method_id, signature)
            validation_details[method_id] = result

            if result["is_valid"]:
                valid_count += 1
            else:
                invalid_count += 1

            if result["missing_fields"]:
                incomplete_count += 1

            if result["warnings"]:
                warnings_count += 1

        # Generate summary statistics
        total_methods = len(methods)
        completeness_rate = (
            (valid_count / total_methods * 100) if total_methods > 0 else 0.0
        )

        # Analyze input patterns
        required_inputs_stats: dict[str, int] = {}
        optional_inputs_stats: dict[str, int] = {}
        critical_optional_stats: dict[str, int] = {}
        output_type_stats: dict[str, int] = {}

        for method_id, method_data in methods.items():
            if "signature" in method_data:
                signature = method_data["signature"]
            else:
                signature = method_data

            # Count required inputs
            for inp in signature.get("required_inputs", []):
                required_inputs_stats[inp] = required_inputs_stats.get(inp, 0) + 1

            # Count optional inputs
            for inp in signature.get("optional_inputs", []):
                optional_inputs_stats[inp] = optional_inputs_stats.get(inp, 0) + 1

            # Count critical optional
            for inp in signature.get("critical_optional", []):
                critical_optional_stats[inp] = critical_optional_stats.get(inp, 0) + 1

            # Count output types
            output_type = signature.get("output_type", "unknown")
            output_type_stats[output_type] = output_type_stats.get(output_type, 0) + 1

        summary = {
            "completeness_rate": round(completeness_rate, 2),
            "methods_with_required_fields": valid_count,
            "methods_missing_required_fields": invalid_count,
            "methods_with_incomplete_signatures": incomplete_count,
            "most_common_required_inputs": sorted(
                required_inputs_stats.items(), key=lambda x: x[1], reverse=True
            )[:5],
            "most_common_optional_inputs": sorted(
                optional_inputs_stats.items(), key=lambda x: x[1], reverse=True
            )[:5],
            "most_common_critical_optional": sorted(
                critical_optional_stats.items(), key=lambda x: x[1], reverse=True
            )[:5],
            "output_type_distribution": output_type_stats,
        }

        return ValidationReport(
            validation_timestamp=datetime.utcnow().isoformat() + "Z",
            signatures_version=self.signatures_data.get(
                "signatures_version", "unknown"
            ),
            total_methods=total_methods,
            valid_methods=valid_count,
            invalid_methods=invalid_count,
            incomplete_methods=incomplete_count,
            methods_with_warnings=warnings_count,
            validation_details=validation_details,
            summary=summary,
        )

    def generate_validation_report(self, output_path: Path | str) -> None:
        """Generate and save validation report to JSON file."""
        report = self.validate_all_signatures()
        output_path = Path(output_path)

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

        print(f"Validation report generated: {output_path}")
        print(f"Total methods: {report['total_methods']}")
        print(f"Valid methods: {report['valid_methods']}")
        print(f"Invalid methods: {report['invalid_methods']}")
        print(f"Completeness rate: {report['summary']['completeness_rate']}%")

    def check_signature_completeness(self, method_id: str) -> bool:
        """
        Check if a method has complete signature with all required fields.

        Returns:
            True if signature has all required fields, False otherwise
        """
        if not self.signatures_data:
            self.load_signatures()

        methods = self.signatures_data.get("methods", {})
        if method_id not in methods:
            return False

        method_data = methods[method_id]
        signature = method_data.get("signature", method_data)

        result = self.validate_signature(method_id, signature)
        return result["is_valid"]

    def get_method_signature(self, method_id: str) -> MethodSignature | None:
        """Retrieve method signature by ID."""
        if not self.signatures_data:
            self.load_signatures()

        methods = self.signatures_data.get("methods", {})
        if method_id not in methods:
            return None

        method_data = methods[method_id]
        if "signature" in method_data:
            return method_data["signature"]
        return method_data


def validate_signatures_cli() -> None:
    """CLI entry point for signature validation."""
    import sys

    signatures_path = "config/json_files_ no_schemas/method_signatures.json"
    output_path = "signature_validation_report.json"

    if len(sys.argv) > 1:
        signatures_path = sys.argv[1]
    if len(sys.argv) > 2:
        output_path = sys.argv[2]

    validator = MethodSignatureValidator(signatures_path)
    validator.generate_validation_report(output_path)


if __name__ == "__main__":
    validate_signatures_cli()



################################################################################
# FILE: phase2_h_metrics_persistence.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_h_metrics_persistence.py
################################################################################

"""
Module: phase2_h_metrics_persistence
PHASE_LABEL: Phase 2
Sequence: H

"""
"""Metrics persistence for PhaseInstrumentation telemetry.

This module provides functions to persist Orchestrator metrics and telemetry
into artifacts/ directory for CI analysis and regression detection.
"""

from __future__ import annotations

import json
from pathlib import Path
from typing import Any
```

Phase 2 Source Code

```
def persist_phase_metrics(
    metrics_data: dict[str, Any],
    output_dir: Path,
    filename: str = "phase_metrics.json"
) -> Path:
    """Persist full PhaseInstrumentation metrics for each phase.

    Args:
        metrics_data: Dictionary containing phase_metrics from export_metrics()
        output_dir: Directory to write metrics files
        filename: Name of the output file

    Returns:
        Path to the written file

    Raises:
        ValueError: If metrics_data is invalid
        OSError: If file cannot be written
    """
    if not isinstance(metrics_data, dict):
        raise ValueError("metrics_data must be a dictionary")

    output_dir.mkdir(parents=True, exist_ok=True)
    output_path = output_dir / filename

    with output_path.open('w', encoding='utf-8') as f:
        json.dump(metrics_data, f, indent=2, sort_keys=True, ensure_ascii=False)

    return output_path


def persist_resource_usage(
    usage_history: list[dict[str, float]],
    output_dir: Path,
    filename: str = "resource_usage.jsonl"
) -> Path:
    """Persist ResourceLimits usage history as JSONL.

    Each line is a JSON object representing a resource usage snapshot.

    Args:
        usage_history: List of usage snapshots from ResourceLimits.get_usage_history()
        output_dir: Directory to write metrics files
        filename: Name of the output file

    Returns:
        Path to the written file

    Raises:
        ValueError: If usage_history is invalid
        OSError: If file cannot be written
    """
    if not isinstance(usage_history, list):
        raise ValueError("usage_history must be a list")

    output_dir.mkdir(parents=True, exist_ok=True)
    output_path = output_dir / filename

    with output_path.open('w', encoding='utf-8') as f:
        for entry in usage_history:
            json.dump(entry, f, ensure_ascii=False)
            f.write('\n')

    return output_path


def persist_latency_histograms(
    phase_metrics: dict[str, Any],
    output_dir: Path,
    filename: str = "latency_histograms.json"
) -> Path:
    """Extract and persist per-phase latency percentiles.

    Args:
        phase_metrics: Dictionary of phase metrics from export_metrics()['phase_metrics']
        output_dir: Directory to write metrics files
        filename: Name of the output file

    Returns:
        Path to the written file

    Raises:
        ValueError: If phase_metrics is invalid
        OSError: If file cannot be written
    """
    if not isinstance(phase_metrics, dict):
        raise ValueError("phase_metrics must be a dictionary")

    output_dir.mkdir(parents=True, exist_ok=True)
    output_path = output_dir / filename

    histograms = {}
    for phase_id, phase_data in phase_metrics.items():
        if isinstance(phase_data, dict) and "latency_histogram" in phase_data:
            histograms[phase_id] = {
                "name": phase_data.get("name", f"phase_{phase_id}"),
                "latency_histogram": phase_data["latency_histogram"],
                "items_processed": phase_data.get("items_processed", 0),
                "duration_ms": phase_data.get("duration_ms"),
                "throughput": phase_data.get("throughput"),
            }

    with output_path.open('w', encoding='utf-8') as f:
        json.dump(histograms, f, indent=2, sort_keys=True, ensure_ascii=False)

    return output_path


def persist_all_metrics(
    orchestrator_metrics: dict[str, Any],
    output_dir: Path
) -> dict[str, Path]:
    """Persist all orchestrator metrics to output directory.

    This is the main entry point for persisting metrics. It writes:
    - phase_metrics.json: Full PhaseInstrumentation.build_metrics() for each phase
    - resource_usage.jsonl: Serialized ResourceLimits.get_usage_history() snapshots
    - latency_histograms.json: Per-phase latency percentiles

    Args:
        orchestrator_metrics: Full metrics dict from Orchestrator.export_metrics()
        output_dir: Directory to write metrics files

    Returns:
        Dictionary mapping metric type to file path

    Raises:
        ValueError: If orchestrator_metrics is invalid
        OSError: If files cannot be written
    """
    if not isinstance(orchestrator_metrics, dict):
        raise ValueError("orchestrator_metrics must be a dictionary")

    phase_metrics = orchestrator_metrics.get("phase_metrics", {})
    resource_usage = orchestrator_metrics.get("resource_usage", [])

    written_files = {}

    written_files["phase_metrics"] = persist_phase_metrics(
        phase_metrics,
        output_dir,
        "phase_metrics.json"
    )

    written_files["resource_usage"] = persist_resource_usage(
        resource_usage,
        output_dir,
        "resource_usage.jsonl"
    )

    written_files["latency_histograms"] = persist_latency_histograms(
        phase_metrics,
        output_dir,
        "latency_histograms.json"
    )

    return written_files


def validate_metrics_schema(metrics_data: dict[str, Any]) -> list[str]:
    """Validate that metrics data conforms to expected schema.

    Args:
        metrics_data: Metrics dictionary from export_metrics()

    Returns:
        List of validation errors (empty if valid)
    """
    errors = []

    if not isinstance(metrics_data, dict):
        errors.append("metrics_data must be a dictionary")
        return errors

    required_keys = ["timestamp", "phase_metrics", "resource_usage", "abort_status", "phase_status"]
    for key in required_keys:
        if key not in metrics_data:
            errors.append(f"Missing required key: {key}")

    if "phase_metrics" in metrics_data:
        if not isinstance(metrics_data["phase_metrics"], dict):
            errors.append("phase_metrics must be a dictionary")
        else:
            for phase_id, phase_data in metrics_data["phase_metrics"].items():
                if not isinstance(phase_data, dict):
                    errors.append(f"phase_metrics[{phase_id}] must be a dictionary")
                    continue

                required_phase_keys = [
                    "phase_id", "name", "duration_ms", "items_processed",
                    "items_total", "latency_histogram"
                ]
                for key in required_phase_keys:
                    if key not in phase_data:
                        errors.append(f"phase_metrics[{phase_id}] missing key: {key}")

    if "resource_usage" in metrics_data and not isinstance(metrics_data["resource_usage"], list):
        errors.append("resource_usage must be a list")

    if "abort_status" in metrics_data and not isinstance(metrics_data["abort_status"], dict):
        errors.append("abort_status must be a dictionary")

    return errors



################################################################################
# FILE: phase2_i_precision_tracking.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_i_precision_tracking.py
################################################################################

"""
Module: phase2_i_precision_tracking
PHASE_LABEL: Phase 2
Sequence: I

"""
"""
Precision Improvement Tracking for Context Filtering
====================================================

Enhanced validation and comprehensive stats tracking for the 60% precision
improvement target from filter_patterns_by_context integration.

This module provides:
1. Enhanced get_patterns_for_context() wrapper with validation
2. Detailed validation status tracking
3. Comprehensive logging and metrics
4. Target achievement verification

Usage:
    >>> from orchestration.orchestrator.precision_tracking import (
    ...     get_patterns_with_validation
    ... )
    >>> patterns, stats = get_patterns_with_validation(
    ...     enriched_pack, document_context
    ... )
    >>> assert stats['integration_validated']
    >>> assert stats['target_achieved']

Author: F.A.R.F.A.N Pipeline
Date: 2025-12-03
"""

from datetime import datetime, timezone
from typing import Any

try:
    import structlog

    logger = structlog.get_logger(__name__)
except ImportError:
    import logging

    logger = logging.getLogger(__name__)


PRECISION_TARGET_THRESHOLD = 0.55


def get_patterns_with_validation(
    enriched_pack: Any,
    document_context: dict[str, Any],
    track_precision_improvement: bool = True,
) -> tuple[list[dict[str, Any]], dict[str, Any]]:
    """
    Enhanced wrapper for get_patterns_for_context() with comprehensive validation.

    This function wraps EnrichedSignalPack.get_patterns_for_context() and adds:
    - Pre-filtering validation
    - Post-filtering verification
    - Integration status checking
    - Target achievement tracking
    - Detailed logging

    Args:
        enriched_pack: EnrichedSignalPack instance
        document_context: Document context dict
        track_precision_improvement: Enable precision tracking

    Returns:
        Tuple of (filtered_patterns, comprehensive_stats) with enhanced fields:
            - validation_timestamp: ISO timestamp
            - validation_details: Detailed validation info
            - target_achieved: Boolean for 60% target
            - validation_status: Status string
            - target_status: Status string
            - pre_filter_count: Patterns before filtering
            - post_filter_count: Patterns after filtering
            - filtering_successful: Boolean validation

    Example:
        >>> enriched = create_enriched_signal_pack(base_pack)
        >>> context = create_document_context(section='budget', chapter=3)
        >>> patterns, stats = get_patterns_with_validation(enriched, context)
        >>> print(f"Validation: {stats['validation_status']}")
        >>> print(f"Target: {stats['target_status']}")
        >>> assert stats['integration_validated']
        >>> assert stats['target_achieved']
    """
    if not isinstance(document_context, dict):
        logger.warning(
            "invalid_document_context_type",
            context_type=type(document_context).__name__,
            expected="dict",
        )
        document_context = {}

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

    pre_filter_count = (
        len(enriched_pack.patterns) if hasattr(enriched_pack, "patterns") else 0
    )

    filtered, base_stats = enriched_pack.get_patterns_for_context(
        document_context, track_precision_improvement=track_precision_improvement
    )

    post_filter_count = len(filtered)

    validation_details = {
        "filter_function_called": True,
        "pre_filter_count": pre_filter_count,
        "post_filter_count": post_filter_count,
        "context_fields": list(document_context.keys()),
        "context_field_count": len(document_context),
        "filtering_successful": post_filter_count <= pre_filter_count,
        "patterns_reduced": pre_filter_count - post_filter_count,
        "reduction_percentage": (
            (pre_filter_count - post_filter_count) / pre_filter_count * 100
            if pre_filter_count > 0
            else 0.0
        ),
    }

    enhanced_stats = {**base_stats}
    enhanced_stats["validation_timestamp"] = validation_timestamp
    enhanced_stats["validation_details"] = validation_details
    enhanced_stats["pre_filter_count"] = pre_filter_count
    enhanced_stats["post_filter_count"] = post_filter_count
    enhanced_stats["filtering_successful"] = validation_details["filtering_successful"]

    if track_precision_improvement:
        integration_validated = base_stats.get("integration_validated", False)
        false_positive_reduction = base_stats.get("false_positive_reduction", 0.0)
        target_achieved = false_positive_reduction >= PRECISION_TARGET_THRESHOLD

        enhanced_stats["target_achieved"] = target_achieved

        if integration_validated:
            enhanced_stats["validation_status"] = "VALIDATED"
            validation_message = "â\234\223 filter_patterns_by_context integration VALIDATED"
        else:
            enhanced_stats["validation_status"] = "NOT_VALIDATED"
            validation_message = (
                "â\234\227 filter_patterns_by_context integration NOT validated"
            )

        target_status = "ACHIEVED" if target_achieved else "NOT_MET"
        enhanced_stats["target_status"] = target_status

        if not validation_details["filtering_successful"]:
            logger.error(
                "context_filtering_validation_failed",
                pre_filter_count=pre_filter_count,
                post_filter_count=post_filter_count,
                reason="filtered_count_exceeds_original",
            )
            enhanced_stats["integration_validated"] = False
            enhanced_stats["validation_status"] = "FAILED"

        logger.info(
            "enhanced_context_filtering_validation",
            pre_filter_count=pre_filter_count,
            post_filter_count=post_filter_count,
            patterns_reduced=validation_details["patterns_reduced"],
            reduction_percentage=f"{validation_details['reduction_percentage']:.1f}%",
            filter_rate=f"{base_stats.get('filter_rate', 0.0):.1%}",
            precision_improvement=f"{base_stats.get('precision_improvement', 0.0):.1%}",
            false_positive_reduction=f"{false_positive_reduction:.1%}",
            integration_validated=integration_validated,
            validation_status=enhanced_stats["validation_status"],
            target_achieved=target_achieved,
            target_status=target_status,
            validation_message=validation_message,
            validation_timestamp=validation_timestamp,
        )

        if target_achieved:
            logger.info(
                "precision_target_achieved",
                false_positive_reduction=f"{false_positive_reduction:.1%}",
                target_threshold=f"{PRECISION_TARGET_THRESHOLD:.1%}",
                message="â\234\223 60% precision improvement target ACHIEVED",
            )
        else:
            logger.warning(
                "precision_target_not_met",
                false_positive_reduction=f"{false_positive_reduction:.1%}",
                target_threshold=f"{PRECISION_TARGET_THRESHOLD:.1%}",
                shortfall=f"{(PRECISION_TARGET_THRESHOLD - false_positive_reduction):.1%}",
                message="â\234\227 60% precision improvement target NOT met",
            )
    else:
        enhanced_stats["target_achieved"] = False
        enhanced_stats["validation_status"] = "TRACKING_DISABLED"
        enhanced_stats["target_status"] = "UNKNOWN"
        logger.debug("context_filtering_applied_without_tracking", **validation_details)

    return filtered, enhanced_stats


def validate_filter_integration(
    enriched_pack: Any, test_contexts: list[dict[str, Any]] | None = None
) -> dict[str, Any]:
    """
    Comprehensive validation of filter_patterns_by_context integration.

    Tests the filtering functionality across multiple contexts and validates:
    - Integration is working correctly
    - Patterns are being filtered
    - 60% target is achievable
    - No errors occur during filtering

    Args:
        enriched_pack: EnrichedSignalPack instance to test
        test_contexts: Optional list of test contexts. If None, uses defaults.

    Returns:
        Validation report dict with:
            - total_tests: Number of contexts tested
            - successful_tests: Tests that completed without error
            - integration_validated: Overall integration status
            - target_achieved_count: Number of tests achieving 60% target
            - target_achievement_rate: Percentage achieving target
            - average_filter_rate: Average pattern reduction
            - average_fp_reduction: Average false positive reduction
            - validation_summary: Human-readable summary

    Example:
        >>> enriched = create_enriched_signal_pack(base_pack)
        >>> report = validate_filter_integration(enriched)
        >>> print(report['validation_summary'])
        >>> assert report['integration_validated']
        >>> assert report['target_achievement_rate'] > 0.5
    """
    if test_contexts is None:
        test_contexts = [
            {},
            {"section": "budget"},
            {"section": "indicators", "chapter": 5},
            {"section": "financial", "chapter": 2, "page": 10},
            {"policy_area": "economic_development"},
        ]

    results = []
    errors = []

    for idx, context in enumerate(test_contexts):
        try:
            patterns, stats = get_patterns_with_validation(
                enriched_pack, context, track_precision_improvement=True
            )
            results.append(stats)
        except Exception as e:
            logger.error(
                "filter_validation_test_failed",
                test_index=idx,
                context=context,
                error=str(e),
                error_type=type(e).__name__,
            )
            errors.append(
                {
                    "test_index": idx,
                    "context": context,
                    "error": str(e),
                    "error_type": type(e).__name__,
                }
            )
    total_tests = len(test_contexts)
    successful_tests = len(results)
    failed_tests = len(errors)

    if successful_tests == 0:
        return {
            "total_tests": total_tests,
            "successful_tests": 0,
            "failed_tests": failed_tests,
            "integration_validated": False,
            "target_achieved_count": 0,
            "target_achievement_rate": 0.0,
            "average_filter_rate": 0.0,
            "average_fp_reduction": 0.0,
            "errors": errors,
            "validation_summary": "â\234\227 ALL TESTS FAILED - Integration NOT working",
        }

    integration_validated_count = sum(
        1 for r in results if r.get("integration_validated", False)
    )
    target_achieved_count = sum(1 for r in results if r.get("target_achieved", False))

    average_filter_rate = (
        sum(r.get("filter_rate", 0.0) for r in results) / successful_tests
    )
    average_fp_reduction = (
        sum(r.get("false_positive_reduction", 0.0) for r in results) / successful_tests
    )

    integration_rate = integration_validated_count / successful_tests
    target_achievement_rate = target_achieved_count / successful_tests

    overall_integration_validated = integration_rate >= 0.8

    validation_summary = (
        f"Filter Integration Validation Report:\n"
        f"  Tests: {successful_tests}/{total_tests} successful ({failed_tests} failed)\n"
        f"  Integration validated: {integration_validated_count}/{successful_tests} "
        f"({integration_rate:.0%})\n"
        f"  60% target achieved: {target_achieved_count}/{successful_tests} "
        f"({target_achievement_rate:.0%})\n"
        f"  Average filter rate: {average_filter_rate:.1%}\n"
        f"  Average FP reduction: {average_fp_reduction:.1%}\n"
        f"  Overall status: "
        f"{'â\234\223 VALIDATED' if overall_integration_validated else 'â\234\227 NOT VALIDATED'}\n"
        f"  Target status: "
        f"{'â\234\223 ACHIEVABLE' if target_achievement_rate > 0 else 'â\234\227 NOT ACHIEVABLE'}"
    )

    report = {
        "total_tests": total_tests,
        "successful_tests": successful_tests,
        "failed_tests": failed_tests,
        "integration_validated": overall_integration_validated,
        "integration_validated_count": integration_validated_count,
        "integration_rate": integration_rate,
        "target_achieved_count": target_achieved_count,
        "target_achievement_rate": target_achievement_rate,
        "average_filter_rate": average_filter_rate,
        "average_fp_reduction": average_fp_reduction,
        "max_fp_reduction": (
            max(r.get("false_positive_reduction", 0.0) for r in results)
            if results
            else 0.0
        ),
        "min_fp_reduction": (
            min(r.get("false_positive_reduction", 0.0) for r in results)
            if results
            else 0.0
        ),
        "errors": errors,
        "validation_summary": validation_summary,
        "all_results": results,
    }

    logger.info(
        "filter_integration_validation_complete",
        total_tests=total_tests,
        successful_tests=successful_tests,
        failed_tests=failed_tests,
        integration_validated=overall_integration_validated,
        target_achievement_rate=f"{target_achievement_rate:.0%}",
        summary=validation_summary,
    )

    return report


def create_precision_tracking_session(
    enriched_pack: Any, session_id: str | None = None
) -> dict[str, Any]:
    """
    Create a precision tracking session for continuous monitoring.
```

Phase 2 Source Code

```
    This creates a session object that tracks multiple measurements over time,
    useful for monitoring precision improvement during production analysis.

    Args:
        enriched_pack: EnrichedSignalPack instance
        session_id: Optional session identifier

    Returns:
        Session object with tracking state and methods

    Example:
        >>> session = create_precision_tracking_session(enriched_pack, "prod_001")
        >>> # Use session throughout analysis...
        >>> results = finalize_precision_tracking_session(session)
    """
    from datetime import datetime, timezone
    from uuid import uuid4

    if session_id is None:
        session_id = f"precision_session_{uuid4().hex[:8]}"

    session = {
        "session_id": session_id,
        "start_timestamp": datetime.now(timezone.utc).isoformat(),
        "enriched_pack": enriched_pack,
        "measurements": [],
        "measurement_count": 0,
        "contexts_tested": [],
        "cumulative_stats": {
            "total_patterns_processed": 0,
            "total_patterns_filtered": 0,
            "total_filtering_time_ms": 0.0,
        },
        "status": "ACTIVE",
    }

    logger.info(
        "precision_tracking_session_created",
        session_id=session_id,
        start_timestamp=session["start_timestamp"],
    )

    return session


def add_measurement_to_session(
    session: dict[str, Any],
    document_context: dict[str, Any],
    track_precision: bool = True,
) -> tuple[list[dict[str, Any]], dict[str, Any]]:
    """
    Add a measurement to an active precision tracking session.

    Args:
        session: Active session from create_precision_tracking_session
        document_context: Document context for this measurement
        track_precision: Enable precision tracking

    Returns:
        Tuple of (filtered_patterns, stats) from get_patterns_for_context

    Example:
        >>> session = create_precision_tracking_session(enriched_pack)
        >>> for context in contexts:
        ...     patterns, stats = add_measurement_to_session(session, context)
    """
    if session["status"] != "ACTIVE":
        logger.warning(
            "measurement_to_inactive_session",
            session_id=session["session_id"],
            status=session["status"],
        )

    enriched_pack = session["enriched_pack"]
    patterns, stats = get_patterns_with_validation(
        enriched_pack, document_context, track_precision
    )

    session["measurements"].append(stats)
    session["measurement_count"] += 1
    session["contexts_tested"].append(document_context)

    session["cumulative_stats"]["total_patterns_processed"] += stats.get(
        "total_patterns", 0
    )
    session["cumulative_stats"]["total_patterns_filtered"] += stats.get(
        "total_patterns", 0
    ) - stats.get("passed", 0)
    session["cumulative_stats"]["total_filtering_time_ms"] += stats.get(
        "filtering_duration_ms", 0.0
    )

    return patterns, stats


def finalize_precision_tracking_session(
    session: dict[str, Any], generate_full_report: bool = True
) -> dict[str, Any]:
    """
    Finalize a precision tracking session and generate summary.

    Args:
        session: Active session to finalize
        generate_full_report: Include full detailed report

    Returns:
        Finalized session report with comprehensive metrics

    Example:
        >>> session = create_precision_tracking_session(enriched_pack)
        >>> # ... add measurements ...
        >>> results = finalize_precision_tracking_session(session)
        >>> print(results['summary'])
    """
    from datetime import datetime, timezone

    from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_intelligence_layer import (
        generate_precision_improvement_report,
    )

    end_timestamp = datetime.now(timezone.utc).isoformat()
    session["end_timestamp"] = end_timestamp
    session["status"] = "FINALIZED"

    if not session["measurements"]:
        return {
            "session_id": session["session_id"],
            "status": "FINALIZED",
            "measurement_count": 0,
            "summary": "No measurements recorded",
        }

    full_report = None
    if generate_full_report:
        full_report = generate_precision_improvement_report(
            session["measurements"], include_detailed_breakdown=True
        )

    session_summary = {
        "session_id": session["session_id"],
        "start_timestamp": session["start_timestamp"],
        "end_timestamp": end_timestamp,
        "status": session["status"],
        "measurement_count": session["measurement_count"],
        "cumulative_stats": session["cumulative_stats"],
        "contexts_tested_count": len(session["contexts_tested"]),
    }

    if full_report:
        session_summary["aggregate_report"] = full_report
        session_summary["summary"] = full_report["summary"]
        session_summary["target_achievement_rate"] = full_report[
            "target_achievement_rate"
        ]
        session_summary["integration_validated"] = full_report["validation_rate"] >= 0.8
        session_summary["validation_health"] = full_report["validation_health"]

    logger.info(
        "precision_tracking_session_finalized",
        session_id=session["session_id"],
        measurement_count=session["measurement_count"],
        total_patterns_processed=session["cumulative_stats"][
            "total_patterns_processed"
        ],
        total_filtering_time_ms=session["cumulative_stats"]["total_filtering_time_ms"],
        target_achievement_rate=(session_summary.get("target_achievement_rate", 0.0)),
    )

    return session_summary


def compare_precision_across_policy_areas(
    policy_area_packs: dict[str, Any], test_contexts: list[dict[str, Any]] | None = None
) -> dict[str, Any]:
    """
    Compare precision improvement across multiple policy areas.

    Useful for identifying which policy areas achieve the 60% target and which need improvement.

    Args:
        policy_area_packs: Dict mapping policy_area_id to EnrichedSignalPack
        test_contexts: Optional test contexts (uses defaults if None)

    Returns:
        Comparison report with per-area metrics and rankings

    Example:
        >>> packs = {
        ...     "PA01": create_enriched_signal_pack(base_pack_01),
        ...     "PA02": create_enriched_signal_pack(base_pack_02),
        ... }
        >>> comparison = compare_precision_across_policy_areas(packs)
        >>> print(comparison['rankings']['by_target_achievement'])
    """
    from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_intelligence_layer import (
        generate_precision_improvement_report,
    )

    if test_contexts is None:
        test_contexts = [
            {},
            {"section": "budget"},
            {"section": "indicators"},
            {"section": "financial"},
        ]

    area_results = {}

    for policy_area_id, enriched_pack in policy_area_packs.items():
        measurements = []
        for context in test_contexts:
            try:
                _, stats = enriched_pack.get_patterns_for_context(
                    context, track_precision_improvement=True
                )
                measurements.append(stats)
            except Exception as e:
                logger.error(
                    "policy_area_precision_test_failed",
                    policy_area=policy_area_id,
                    context=context,
                    error=str(e),
                )

        if measurements:
            report = generate_precision_improvement_report(
                measurements, include_detailed_breakdown=False
            )
            area_results[policy_area_id] = report

    if not area_results:
        return {
            "policy_areas_tested": 0,
            "comparison_status": "FAILED",
            "message": "No successful measurements",
        }

    rankings = {
        "by_target_achievement": sorted(
            area_results.items(),
            key=lambda x: x[1]["target_achievement_rate"],
            reverse=True,
        ),
        "by_avg_fp_reduction": sorted(
            area_results.items(),
            key=lambda x: x[1]["avg_false_positive_reduction"],
            reverse=True,
        ),
        "by_validation_rate": sorted(
            area_results.items(), key=lambda x: x[1]["validation_rate"], reverse=True
        ),
    }

    best_performer = rankings["by_target_achievement"][0]
    worst_performer = rankings["by_target_achievement"][-1]

    areas_meeting_target = sum(
        1
        for _, report in area_results.items()
        if report["max_false_positive_reduction"] >= PRECISION_TARGET_THRESHOLD
    )

    comparison_summary = (
        f"Policy Area Precision Comparison:\n"
        f"  Areas tested: {len(area_results)}\n"
        f"  Areas meeting 60% target: {areas_meeting_target}/{len(area_results)}\n"
        f"  Best performer: {best_performer[0]} "
        f"({100*best_performer[1]['target_achievement_rate']:.0f}% target achievement)\n"
        f"  Worst performer: {worst_performer[0]} "
        f"({100*worst_performer[1]['target_achievement_rate']:.0f}% target achievement)\n"
        f"  Overall status: "
        f"{'â\234\223 GOOD' if areas_meeting_target >= len(area_results) * 0.7 else 'â\234\227 NEEDS IMPROVEMENT'}"
    )

    return {
        "policy_areas_tested": len(area_results),
        "areas_meeting_target": areas_meeting_target,
        "target_achievement_coverage": areas_meeting_target / len(area_results),
        "rankings": rankings,
        "best_performer": {
            "policy_area": best_performer[0],
            "metrics": best_performer[1],
        },
        "worst_performer": {
            "policy_area": worst_performer[0],
            "metrics": worst_performer[1],
        },
        "all_results": area_results,
        "comparison_summary": comparison_summary,
    }


def export_precision_metrics_for_monitoring(
    measurements: list[dict[str, Any]], output_format: str = "json"
) -> str | dict[str, Any]:
    """
    Export precision metrics in format suitable for external monitoring systems.

    Args:
        measurements: List of stats dicts from get_patterns_for_context
        output_format: 'json', 'prometheus', or 'datadog'

    Returns:
        Formatted metrics string or dict

    Example:
        >>> measurements = [...]
        >>> metrics = export_precision_metrics_for_monitoring(measurements, 'json')
    """
    import json
    from datetime import datetime, timezone

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

    if not measurements:
        if output_format == "json":
            return json.dumps({"error": "No measurements", "timestamp": timestamp})
        return ""

    total = len(measurements)
    meets_target = sum(
        1
        for m in measurements
        if m.get("false_positive_reduction", 0.0) >= PRECISION_TARGET_THRESHOLD
    )
    validated = sum(1 for m in measurements if m.get("integration_validated", False))

    avg_fp_reduction = (
        sum(m.get("false_positive_reduction", 0.0) for m in measurements) / total
    )
    avg_filter_rate = sum(m.get("filter_rate", 0.0) for m in measurements) / total

    if output_format == "json":
        return json.dumps(
            {
                "timestamp": timestamp,
                "measurement_count": total,
                "target_achievement_count": meets_target,
                "target_achievement_rate": meets_target / total,
                "integration_validated_count": validated,
                "integration_validation_rate": validated / total,
                "avg_false_positive_reduction": avg_fp_reduction,
                "avg_filter_rate": avg_filter_rate,
                "meets_60_percent_target": meets_target / total >= 0.5,
            },
            indent=2,
        )

    elif output_format == "prometheus":
        lines = [
            "# HELP precision_target_achievement_rate Rate of measurements meeting 60% target",
            "# TYPE precision_target_achievement_rate gauge",
            f"precision_target_achievement_rate {meets_target / total}",
            "# HELP precision_avg_fp_reduction Average false positive reduction",
            "# TYPE precision_avg_fp_reduction gauge",
            f"precision_avg_fp_reduction {avg_fp_reduction}",
            "# HELP precision_measurement_count Total measurements",
            "# TYPE precision_measurement_count counter",
            f"precision_measurement_count {total}",
        ]
        return "\n".join(lines)

    elif output_format == "datadog":
        return json.dumps(
            [
                {
                    "metric": "farfan.precision.target_achievement_rate",
                    "points": [
                        [
                            int(datetime.now(timezone.utc).timestamp()),
                            meets_target / total,
                        ]
                    ],
                    "type": "gauge",
                    "tags": ["component:context_filtering"],
                },
                {
                    "metric": "farfan.precision.avg_fp_reduction",
                    "points": [
                        [int(datetime.now(timezone.utc).timestamp()), avg_fp_reduction]
                    ],
                    "type": "gauge",
                    "tags": ["component:context_filtering"],
                },
                {
                    "metric": "farfan.precision.measurement_count",
                    "points": [[int(datetime.now(timezone.utc).timestamp()), total]],
                    "type": "count",
                    "tags": ["component:context_filtering"],
                },
            ],
            indent=2,
        )

    return ""


__all__ = [
    "get_patterns_with_validation",
    "validate_filter_integration",
    "create_precision_tracking_session",
    "add_measurement_to_session",
    "finalize_precision_tracking_session",
    "compare_precision_across_policy_areas",
    "export_precision_metrics_for_monitoring",
    "PRECISION_TARGET_THRESHOLD",
]



################################################################################
# FILE: phase2_j_resource_integration.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_j_resource_integration.py
################################################################################

"""
Module: phase2_j_resource_integration
PHASE_LABEL: Phase 2
Sequence: J

"""
"""Resource Management Integration.

Factory functions and helpers to integrate adaptive resource management
with the existing orchestrator infrastructure.
"""

from __future__ import annotations

import logging
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor, Orchestrator, ResourceLimits

from orchestration.resource_alerts import (
    AlertChannel,
    AlertThresholds,
    ResourceAlertManager,
)
from orchestration.resource_aware_executor import ResourceAwareExecutor
from orchestration.resource_manager import (
    AdaptiveResourceManager,
    ExecutorPriority,
    ResourceAllocationPolicy,
)

logger = logging.getLogger(__name__)


def create_resource_manager(
    resource_limits: ResourceLimits,
    enable_circuit_breakers: bool = True,
    enable_degradation: bool = True,
    enable_alerts: bool = True,
    alert_channels: list[AlertChannel] | None = None,
    alert_webhook_url: str | None = None,
) -> tuple[AdaptiveResourceManager, ResourceAlertManager | None]:
    """Create and configure adaptive resource manager with alerts.

    Args:
        resource_limits: Existing ResourceLimits instance
        enable_circuit_breakers: Enable circuit breaker protection
        enable_degradation: Enable graceful degradation
        enable_alerts: Enable alerting system
        alert_channels: Alert delivery channels
        alert_webhook_url: Webhook URL for external alerts

    Returns:
        Tuple of (AdaptiveResourceManager, ResourceAlertManager)
    """
    alert_manager = None

    if enable_alerts:
        thresholds = AlertThresholds(
            memory_warning_percent=75.0,
            memory_critical_percent=85.0,
            cpu_warning_percent=75.0,
            cpu_critical_percent=85.0,
            circuit_breaker_warning_count=3,
            degradation_critical_count=3,
        )

        alert_manager = ResourceAlertManager(
            thresholds=thresholds,
            channels=alert_channels or [AlertChannel.LOG],
            webhook_url=alert_webhook_url,
        )

        alert_callback = alert_manager.process_event
    else:
        alert_callback = None

    resource_manager = AdaptiveResourceManager(
        resource_limits=resource_limits,
        enable_circuit_breakers=enable_circuit_breakers,
        enable_degradation=enable_degradation,
        alert_callback=alert_callback,
    )

    register_default_policies(resource_manager)

    logger.info(
        "Resource management system initialized",
        extra={
            "circuit_breakers": enable_circuit_breakers,
            "degradation": enable_degradation,
            "alerts": enable_alerts,
        },
    )

    return resource_manager, alert_manager


def register_default_policies(
    resource_manager: AdaptiveResourceManager,
) -> None:
    """Register default resource allocation policies for critical executors."""
    policies = [
        ResourceAllocationPolicy(
            executor_id="D3-Q3",
            priority=ExecutorPriority.CRITICAL,
            min_memory_mb=256.0,
            max_memory_mb=1024.0,
            min_workers=2,
            max_workers=8,
            is_memory_intensive=True,
        ),
        ResourceAllocationPolicy(
            executor_id="D4-Q2",
            priority=ExecutorPriority.CRITICAL,
            min_memory_mb=256.0,
            max_memory_mb=1024.0,
            min_workers=2,
            max_workers=8,
            is_memory_intensive=True,
        ),
        ResourceAllocationPolicy(
            executor_id="D3-Q2",
            priority=ExecutorPriority.HIGH,
            min_memory_mb=128.0,
            max_memory_mb=512.0,
            min_workers=1,
            max_workers=6,
        ),
        ResourceAllocationPolicy(
            executor_id="D4-Q1",
            priority=ExecutorPriority.HIGH,
            min_memory_mb=128.0,
            max_memory_mb=512.0,
            min_workers=1,
            max_workers=6,
        ),
        ResourceAllocationPolicy(
            executor_id="D2-Q3",
            priority=ExecutorPriority.HIGH,
            min_memory_mb=128.0,
            max_memory_mb=512.0,
            min_workers=1,
            max_workers=6,
            is_cpu_intensive=True,
        ),
        ResourceAllocationPolicy(
            executor_id="D1-Q1",
            priority=ExecutorPriority.NORMAL,
            min_memory_mb=64.0,
            max_memory_mb=256.0,
            min_workers=1,
            max_workers=4,
        ),
        ResourceAllocationPolicy(
            executor_id="D1-Q2",
            priority=ExecutorPriority.NORMAL,
            min_memory_mb=64.0,
            max_memory_mb=256.0,
            min_workers=1,
            max_workers=4,
        ),
        ResourceAllocationPolicy(
            executor_id="D5-Q1",
            priority=ExecutorPriority.NORMAL,
            min_memory_mb=128.0,
            max_memory_mb=384.0,
            min_workers=1,
            max_workers=4,
        ),
        ResourceAllocationPolicy(
            executor_id="D6-Q1",
            priority=ExecutorPriority.NORMAL,
            min_memory_mb=128.0,
            max_memory_mb=384.0,
            min_workers=1,
            max_workers=4,
        ),
    ]

    for policy in policies:
        resource_manager.register_allocation_policy(policy)


def wrap_method_executor(
    method_executor: MethodExecutor,
    resource_manager: AdaptiveResourceManager,
) -> ResourceAwareExecutor:
    """Wrap MethodExecutor with resource management.

    Args:
        method_executor: Existing MethodExecutor instance
        resource_manager: Configured AdaptiveResourceManager

    Returns:
        ResourceAwareExecutor wrapping the method executor
    """
    return ResourceAwareExecutor(
        method_executor=method_executor,
        resource_manager=resource_manager,
    )


def integrate_with_orchestrator(
    orchestrator: Orchestrator,
    enable_circuit_breakers: bool = True,
    enable_degradation: bool = True,
    enable_alerts: bool = True,
) -> dict[str, Any]:
    """Integrate resource management with existing Orchestrator.

    Args:
        orchestrator: Existing Orchestrator instance
        enable_circuit_breakers: Enable circuit breaker protection
        enable_degradation: Enable graceful degradation
        enable_alerts: Enable alerting system

    Returns:
        Dictionary with resource management components
    """
    if not hasattr(orchestrator, "resource_limits"):
        raise RuntimeError(
            "Orchestrator must have resource_limits attribute"
        )

    resource_manager, alert_manager = create_resource_manager(
        resource_limits=orchestrator.resource_limits,
        enable_circuit_breakers=enable_circuit_breakers,
        enable_degradation=enable_degradation,
        enable_alerts=enable_alerts,
    )

    setattr(orchestrator, "_resource_manager", resource_manager)
    setattr(orchestrator, "_alert_manager", alert_manager)

    logger.info("Resource management integrated with orchestrator")

    return {
        "resource_manager": resource_manager,
        "alert_manager": alert_manager,
        "resource_limits": orchestrator.resource_limits,
    }


def get_resource_status(orchestrator: Orchestrator) -> dict[str, Any]:
    """Get comprehensive resource management status from orchestrator.

    Args:
        orchestrator: Orchestrator with integrated resource management

    Returns:
        Complete resource management status
    """
    status: dict[str, Any] = {
        "resource_management_enabled": False,
        "resource_limits": {},
        "resource_manager": {},
        "alerts": {},
    }

    if hasattr(orchestrator, "resource_limits"):
        status["resource_limits"] = {
            "max_memory_mb": orchestrator.resource_limits.max_memory_mb,
            "max_cpu_percent": orchestrator.resource_limits.max_cpu_percent,
            "max_workers": orchestrator.resource_limits.max_workers,
            "current_usage": orchestrator.resource_limits.get_resource_usage(),
        }

    if hasattr(orchestrator, "_resource_manager"):
        status["resource_management_enabled"] = True
        status["resource_manager"] = (
            orchestrator._resource_manager.get_resource_status()
        )

    if hasattr(orchestrator, "_alert_manager") and orchestrator._alert_manager:
        status["alerts"] = orchestrator._alert_manager.get_alert_summary()

    return status


def reset_circuit_breakers(orchestrator: Orchestrator) -> dict[str, bool]:
    """Reset all circuit breakers in orchestrator.

    Args:
        orchestrator: Orchestrator with integrated resource management

    Returns:
        Dictionary mapping executor_id to reset success status
    """
    if not hasattr(orchestrator, "_resource_manager"):
        return {}

    resource_manager = orchestrator._resource_manager
    results = {}

    for executor_id in resource_manager.circuit_breakers:
        success = resource_manager.reset_circuit_breaker(executor_id)
        results[executor_id] = success

        if success:
            logger.info(f"Reset circuit breaker for {executor_id}")

    return results



################################################################################
# FILE: phase2_k_resource_aware_executor.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_k_resource_aware_executor.py
################################################################################

"""
Module: phase2_k_resource_aware_executor
PHASE_LABEL: Phase 2
Sequence: K

"""
"""Resource-Aware Executor Wrapper.

Integrates AdaptiveResourceManager with MethodExecutor to provide:
- Automatic resource allocation before execution
- Circuit breaker checks before execution
- Degradation configuration injection
- Execution metrics tracking
- Memory and timing instrumentation
"""

from __future__ import annotations

import asyncio
import logging
import time
from typing import TYPE_CHECKING, Any

if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor
    from orchestration.resource_manager import AdaptiveResourceManager

logger = logging.getLogger(__name__)


class ResourceAwareExecutor:
    """Wraps MethodExecutor with adaptive resource management."""

    def __init__(
        self,
        method_executor: MethodExecutor,
        resource_manager: AdaptiveResourceManager,
    ) -> None:
        self.method_executor = method_executor
        self.resource_manager = resource_manager
```

Phase 2 Source Code

```
    async def execute_with_resource_management(
        self,
        executor_id: str,
        context: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Execute with full resource management integration.

        Args:
            executor_id: Executor identifier (e.g., "D3-Q3")
            context: Execution context
            **kwargs: Additional arguments for execution

        Returns:
            Execution result with resource metadata

        Raises:
            RuntimeError: If circuit breaker is open or execution fails
        """
        can_execute, reason = self.resource_manager.can_execute(executor_id)
        if not can_execute:
            logger.warning(
                f"Executor {executor_id} blocked by circuit breaker: {reason}"
            )
            raise RuntimeError(
                f"Executor {executor_id} unavailable: {reason}"
            )

        allocation = await self.resource_manager.start_executor_execution(
            executor_id
        )

        degradation_config = allocation["degradation"]
        enriched_context = self._apply_degradation(context, degradation_config)

        logger.info(
            f"Executing {executor_id} with resource allocation",
            extra={
                "max_memory_mb": allocation["max_memory_mb"],
                "max_workers": allocation["max_workers"],
                "priority": allocation["priority"],
                "degradation_applied": degradation_config["applied_strategies"],
            },
        )

        start_time = time.perf_counter()
        success = False
        result = None
        error = None

        try:
            result = await self._execute_with_timeout(
                executor_id, enriched_context, allocation, **kwargs
            )
            success = True
            return result
        except Exception as exc:
            error = str(exc)
            logger.error(
                f"Executor {executor_id} failed: {exc}",
                exc_info=True,
            )
            raise
        finally:
            duration_ms = (time.perf_counter() - start_time) * 1000

            memory_mb = self._estimate_memory_usage()

            await self.resource_manager.end_executor_execution(
                executor_id=executor_id,
                success=success,
                duration_ms=duration_ms,
                memory_mb=memory_mb,
            )

            logger.info(
                f"Executor {executor_id} completed",
                extra={
                    "success": success,
                    "duration_ms": duration_ms,
                    "memory_mb": memory_mb,
                    "error": error,
                },
            )

    async def _execute_with_timeout(
        self,
        executor_id: str,
        context: dict[str, Any],
        allocation: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Execute with timeout based on resource allocation."""
        timeout_seconds = self._calculate_timeout(allocation)

        try:
            result = await asyncio.wait_for(
                self._execute_async(executor_id, context, **kwargs),
                timeout=timeout_seconds,
            )
            return result
        except asyncio.TimeoutError as exc:
            logger.error(
                f"Executor {executor_id} timed out after {timeout_seconds}s"
            )
            raise RuntimeError(
                f"Executor {executor_id} timed out"
            ) from exc

    async def _execute_async(
        self,
        executor_id: str,
        context: dict[str, Any],
        **kwargs: Any,
    ) -> dict[str, Any]:
        """Async wrapper for executor execution."""
        loop = asyncio.get_event_loop()
        return await loop.run_in_executor(
            None, self._execute_sync, executor_id, context, kwargs
        )

    def _execute_sync(
        self,
        executor_id: str,
        context: dict[str, Any],
        kwargs: dict[str, Any],
    ) -> dict[str, Any]:
        """Synchronous execution wrapper."""
        try:
            from farfan_pipeline.phases.Phase_two.phase2_b_base_executor_with_contract import DynamicContractExecutor

            # Extract question_id from context or executor_id
            # executor_id format could be "D3-Q3" but we need question_id like "Q013"
            question_id = context.get("question_id")
            if not question_id:
                # Try to derive from executor_id if it’s in base_slot format
                if executor_id and "-" in executor_id:
                    # D3-Q3 â\206\222 dimension 3, question 3 â\206\222 Q013
                    parts = executor_id.split("-")
                    if len(parts) == 2 and parts[0].startswith("D") and parts[1].startswith("Q"):
                        dim = int(parts[0][1:])
                        q = int(parts[1][1:])
                        q_number = (dim - 1) * 5 + q
                        question_id = f"Q{q_number:03d}"

            if not question_id:
                raise ValueError(f"Cannot determine question_id from executor_id: {executor_id}")

            # Create GenericContractExecutor with question_id
            # TODO: ResourceAwareExecutor needs update to support BaseExecutorWithContract dependencies.
            # Currently missing signal_registry, config, questionnaire_provider.
            # Bypassing execution for now to maintain structure integrity.
            raise NotImplementedError("ResourceAwareExecutor update pending for Contract-Based Executors")

        except Exception as exc:
            logger.error(f"Sync execution failed: {exc}")
            raise

    def _apply_degradation(
        self,
        context: dict[str, Any],
        degradation_config: dict[str, Any],
    ) -> dict[str, Any]:
        """Apply degradation strategies to context."""
        enriched = context.copy()

        enriched["_resource_constraints"] = {
            "entity_limit_factor": degradation_config["entity_limit_factor"],
            "disable_expensive_computations": degradation_config[
                "disable_expensive_computations"
            ],
            "use_simplified_methods": degradation_config["use_simplified_methods"],
            "skip_optional_analysis": degradation_config["skip_optional_analysis"],
            "reduce_embedding_dims": degradation_config["reduce_embedding_dims"],
        }

        if degradation_config["entity_limit_factor"] < 1.0:
            for key in ["max_entities", "max_chunks", "max_results"]:
                if key in enriched:
                    enriched[key] = int(
                        enriched[key] * degradation_config["entity_limit_factor"]
                    )

        return enriched

    def _calculate_timeout(self, allocation: dict[str, Any]) -> float:
        """Calculate execution timeout based on allocation."""
        base_timeout = 300.0

        priority = allocation["priority"]
        if priority == 1:
            return base_timeout * 1.5
        elif priority == 2:
            return base_timeout * 1.2
        else:
            return base_timeout

    def _estimate_memory_usage(self) -> float:
        """Estimate current memory usage."""
        try:
            import psutil
            process = psutil.Process()
            return process.memory_info().rss / (1024 * 1024)
        except Exception:
            usage = self.resource_manager.resource_limits.get_resource_usage()
            return usage.get("rss_mb", 0.0)


class ResourceConstraints:
    """Helper to extract and apply resource constraints in executors."""

    @staticmethod
    def get_constraints(context: dict[str, Any]) -> dict[str, Any]:
        """Extract resource constraints from context."""
        return context.get(
            "_resource_constraints",
            {
                "entity_limit_factor": 1.0,
                "disable_expensive_computations": False,
                "use_simplified_methods": False,
                "skip_optional_analysis": False,
                "reduce_embedding_dims": False,
            },
        )

    @staticmethod
    def should_skip_expensive_computation(context: dict[str, Any]) -> bool:
        """Check if expensive computations should be skipped."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("disable_expensive_computations", False)

    @staticmethod
    def should_use_simplified_methods(context: dict[str, Any]) -> bool:
        """Check if simplified methods should be used."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("use_simplified_methods", False)

    @staticmethod
    def should_skip_optional_analysis(context: dict[str, Any]) -> bool:
        """Check if optional analysis should be skipped."""
        constraints = ResourceConstraints.get_constraints(context)
        return constraints.get("skip_optional_analysis", False)

    @staticmethod
    def get_entity_limit(context: dict[str, Any], default: int) -> int:
        """Get entity limit with degradation applied."""
        constraints = ResourceConstraints.get_constraints(context)
        factor = constraints.get("entity_limit_factor", 1.0)
        return int(default * factor)

    @staticmethod
    def get_embedding_dimensions(context: dict[str, Any], default: int) -> int:
        """Get embedding dimensions with degradation applied."""
        constraints = ResourceConstraints.get_constraints(context)
        if constraints.get("reduce_embedding_dims", False):
            return int(default * 0.5)
        return default



################################################################################
# FILE: phase2_l_resource_manager.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_l_resource_manager.py
################################################################################

"""
Module: phase2_l_resource_manager
PHASE_LABEL: Phase 2
Sequence: L

"""
"""Adaptive Resource Management System.

Provides dynamic resource allocation, degradation strategies, circuit breakers,
and priority-based resource allocation for policy analysis executors.

This module integrates with ResourceLimits to provide:
- Real-time resource monitoring and adaptive allocation
- Graceful degradation strategies when resources are constrained
- Circuit breakers for memory-intensive executors
- Priority-based resource allocation (critical executors first)
- Comprehensive observability with alerts
"""

from __future__ import annotations

import asyncio
import logging
```

Phase 2 Source Code

```
import time
from collections import defaultdict, deque
from dataclasses import dataclass, field
from datetime import datetime
from enum import Enum
from typing import TYPE_CHECKING, Any, Callable

if TYPE_CHECKING:
    from orchestration.orchestrator import ResourceLimits

logger = logging.getLogger(__name__)


class ResourcePressureLevel(Enum):
    """Resource pressure severity levels."""

    NORMAL = "normal"
    ELEVATED = "elevated"
    HIGH = "high"
    CRITICAL = "critical"
    EMERGENCY = "emergency"


class ExecutorPriority(Enum):
    """Priority levels for executor resource allocation."""

    CRITICAL = 1
    HIGH = 2
    NORMAL = 3
    LOW = 4


class CircuitState(Enum):
    """Circuit breaker states."""

    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"


@dataclass
class ExecutorMetrics:
    """Metrics for individual executor performance and resource usage."""

    executor_id: str
    total_executions: int = 0
    successful_executions: int = 0
    failed_executions: int = 0
    avg_memory_mb: float = 0.0
    peak_memory_mb: float = 0.0
    avg_cpu_percent: float = 0.0
    avg_duration_ms: float = 0.0
    last_execution_time: datetime | None = None
    memory_samples: list[float] = field(default_factory=list)
    duration_samples: list[float] = field(default_factory=list)


@dataclass
class CircuitBreakerConfig:
    """Configuration for circuit breaker behavior."""

    failure_threshold: int = 5
    timeout_seconds: float = 60.0
    half_open_timeout: float = 30.0
    memory_threshold_mb: float = 2048.0
    success_threshold: int = 3


@dataclass
class CircuitBreaker:
    """Circuit breaker for memory-intensive executors."""

    executor_id: str
    state: CircuitState = CircuitState.CLOSED
    failure_count: int = 0
    success_count: int = 0
    last_failure_time: datetime | None = None
    last_state_change: datetime | None = None
    config: CircuitBreakerConfig = field(default_factory=CircuitBreakerConfig)

    def can_execute(self) -> bool:
        """Check if executor can be executed based on circuit state."""
        if self.state == CircuitState.CLOSED:
            return True

        if self.state == CircuitState.OPEN:
            if self.last_state_change:
                elapsed = (datetime.utcnow() - self.last_state_change).total_seconds()
                if elapsed >= self.config.timeout_seconds:
                    self.state = CircuitState.HALF_OPEN
                    self.success_count = 0
                    logger.info(
                        f"Circuit breaker for {self.executor_id} moved to HALF_OPEN"
                    )
                    return True
            return False
        return True

    def record_success(self) -> None:
        """Record successful execution."""
        self.failure_count = 0

        if self.state == CircuitState.HALF_OPEN:
            self.success_count += 1
            if self.success_count >= self.config.success_threshold:
                self.state = CircuitState.CLOSED
                self.last_state_change = datetime.utcnow()
                logger.info(
                    f"Circuit breaker for {self.executor_id} closed after "
                    f"{self.success_count} successes"
                )

    def record_failure(self, memory_mb: float | None = None) -> None:
        """Record failed execution."""
        self.failure_count += 1
        self.last_failure_time = datetime.utcnow()

        exceeded_memory = (
            memory_mb is not None and memory_mb > self.config.memory_threshold_mb
        )

        if self.state == CircuitState.HALF_OPEN:
            self.state = CircuitState.OPEN
            self.last_state_change = datetime.utcnow()
            logger.warning(
                f"Circuit breaker for {self.executor_id} opened from HALF_OPEN "
                f"(memory: {memory_mb}MB)"
            )
        elif (
            self.failure_count >= self.config.failure_threshold or exceeded_memory
        ):
            self.state = CircuitState.OPEN
            self.last_state_change = datetime.utcnow()
            logger.warning(
                f"Circuit breaker for {self.executor_id} opened "
                f"(failures: {self.failure_count}, memory: {memory_mb}MB)"
            )


@dataclass
class DegradationStrategy:
    """Defines degradation behavior for resource-constrained scenarios."""

    name: str
    pressure_threshold: ResourcePressureLevel
    enabled: bool = True
    entity_limit_factor: float = 1.0
    disable_expensive_computations: bool = False
    use_simplified_methods: bool = False
    skip_optional_analysis: bool = False
    reduce_embedding_dims: bool = False
    applied_count: int = 0

    def should_apply(self, pressure: ResourcePressureLevel) -> bool:
        """Check if strategy should be applied at current pressure level."""
        if not self.enabled:
            return False

        pressure_values = {
            ResourcePressureLevel.NORMAL: 0,
            ResourcePressureLevel.ELEVATED: 1,
            ResourcePressureLevel.HIGH: 2,
            ResourcePressureLevel.CRITICAL: 3,
            ResourcePressureLevel.EMERGENCY: 4,
        }

        return pressure_values[pressure] >= pressure_values[self.pressure_threshold]


@dataclass
class ResourceAllocationPolicy:
    """Defines resource allocation priority for executors."""

    executor_id: str
    priority: ExecutorPriority
    min_memory_mb: float
    max_memory_mb: float
    min_workers: int
    max_workers: int
    is_memory_intensive: bool = False
    is_cpu_intensive: bool = False


@dataclass
class ResourcePressureEvent:
    """Event capturing resource pressure state changes."""

    timestamp: datetime
    pressure_level: ResourcePressureLevel
    cpu_percent: float
    memory_mb: float
    memory_percent: float
    worker_count: int
    active_executors: int
    degradation_applied: list[str]
    circuit_breakers_open: list[str]
    message: str


class AdaptiveResourceManager:
    """Manages dynamic resource allocation and degradation strategies."""

    CRITICAL_EXECUTORS = {
        "D3-Q3": ExecutorPriority.CRITICAL,
        "D4-Q2": ExecutorPriority.CRITICAL,
        "D3-Q2": ExecutorPriority.HIGH,
        "D4-Q1": ExecutorPriority.HIGH,
        "D2-Q3": ExecutorPriority.HIGH,
    }

    DEFAULT_POLICIES = {
        "D3-Q3": ResourceAllocationPolicy(
            executor_id="D3-Q3",
            priority=ExecutorPriority.CRITICAL,
            min_memory_mb=256.0,
            max_memory_mb=1024.0,
            min_workers=2,
            max_workers=8,
            is_memory_intensive=True,
        ),
        "D4-Q2": ResourceAllocationPolicy(
            executor_id="D4-Q2",
            priority=ExecutorPriority.CRITICAL,
            min_memory_mb=256.0,
            max_memory_mb=1024.0,
            min_workers=2,
            max_workers=8,
            is_memory_intensive=True,
        ),
        "D3-Q2": ResourceAllocationPolicy(
            executor_id="D3-Q2",
            priority=ExecutorPriority.HIGH,
            min_memory_mb=128.0,
            max_memory_mb=512.0,
            min_workers=1,
            max_workers=6,
        ),
        "D4-Q1": ResourceAllocationPolicy(
            executor_id="D4-Q1",
            priority=ExecutorPriority.HIGH,
            min_memory_mb=128.0,
            max_memory_mb=512.0,
            min_workers=1,
            max_workers=6,
        ),
    }

    def __init__(
        self,
        resource_limits: ResourceLimits,
        enable_circuit_breakers: bool = True,
        enable_degradation: bool = True,
        alert_callback: Callable[[ResourcePressureEvent], None] | None = None,
    ) -> None:
        self.resource_limits = resource_limits
        self.enable_circuit_breakers = enable_circuit_breakers
        self.enable_degradation = enable_degradation
        self.alert_callback = alert_callback

        self.executor_metrics: dict[str, ExecutorMetrics] = {}
        self.circuit_breakers: dict[str, CircuitBreaker] = {}
        self.allocation_policies: dict[str, ResourceAllocationPolicy] = (
            self.DEFAULT_POLICIES.copy()
        )

        self.degradation_strategies = self._init_degradation_strategies()
        self.pressure_history: deque[ResourcePressureEvent] = deque(maxlen=100)
        self.current_pressure = ResourcePressureLevel.NORMAL

        self._lock = asyncio.Lock()
        self._active_executors: set[str] = set()

        logger.info("Adaptive Resource Manager initialized")

    def _init_degradation_strategies(self) -> list[DegradationStrategy]:
        """Initialize degradation strategies for different pressure levels."""
        return [
            DegradationStrategy(
                name="reduce_entity_limits",
                pressure_threshold=ResourcePressureLevel.ELEVATED,
                entity_limit_factor=0.8,
            ),
            DegradationStrategy(
                name="skip_optional_analysis",
                pressure_threshold=ResourcePressureLevel.HIGH,
                skip_optional_analysis=True,
            ),
            DegradationStrategy(
                name="disable_expensive_computations",
                pressure_threshold=ResourcePressureLevel.HIGH,
                disable_expensive_computations=True,
            ),
            DegradationStrategy(
                name="use_simplified_methods",
                pressure_threshold=ResourcePressureLevel.CRITICAL,
                use_simplified_methods=True,
                entity_limit_factor=0.5,
            ),
            DegradationStrategy(
                name="reduce_embedding_dimensions",
                pressure_threshold=ResourcePressureLevel.CRITICAL,
                reduce_embedding_dims=True,
            ),
            DegradationStrategy(
                name="emergency_mode",
                pressure_threshold=ResourcePressureLevel.EMERGENCY,
                entity_limit_factor=0.3,
                disable_expensive_computations=True,
                use_simplified_methods=True,
                skip_optional_analysis=True,
                reduce_embedding_dims=True,
            ),
        ]

    def get_or_create_circuit_breaker(
        self, executor_id: str
    ) -> CircuitBreaker:
        """Get or create circuit breaker for executor."""
        if executor_id not in self.circuit_breakers:
            config = CircuitBreakerConfig()

            if executor_id in self.allocation_policies:
                policy = self.allocation_policies[executor_id]
                if policy.is_memory_intensive:
                    config.memory_threshold_mb = policy.max_memory_mb * 1.5

            self.circuit_breakers[executor_id] = CircuitBreaker(
                executor_id=executor_id, config=config
            )

        return self.circuit_breakers[executor_id]

    def can_execute(self, executor_id: str) -> tuple[bool, str]:
        """Check if executor can be executed based on circuit breaker state."""
        if not self.enable_circuit_breakers:
            return True, "Circuit breakers disabled"

        breaker = self.get_or_create_circuit_breaker(executor_id)

        if not breaker.can_execute():
            return False, f"Circuit breaker is {breaker.state.value}"

        return True, "OK"

    async def assess_resource_pressure(self) -> ResourcePressureLevel:
        """Assess current resource pressure level."""
        usage = self.resource_limits.get_resource_usage()

        cpu_percent = usage.get("cpu_percent", 0.0)
        memory_percent = usage.get("memory_percent", 0.0)
        rss_mb = usage.get("rss_mb", 0.0)

        max_memory_mb = self.resource_limits.max_memory_mb or 4096.0
        max_cpu = self.resource_limits.max_cpu_percent

        memory_ratio = rss_mb / max_memory_mb
        cpu_ratio = cpu_percent / max_cpu if max_cpu else 0.0

        if memory_ratio >= 0.95 or cpu_ratio >= 0.95:
            pressure = ResourcePressureLevel.EMERGENCY
        elif memory_ratio >= 0.85 or cpu_ratio >= 0.85:
            pressure = ResourcePressureLevel.CRITICAL
        elif memory_ratio >= 0.75 or cpu_ratio >= 0.75:
            pressure = ResourcePressureLevel.HIGH
        elif memory_ratio >= 0.65 or cpu_ratio >= 0.65:
            pressure = ResourcePressureLevel.ELEVATED
        else:
            pressure = ResourcePressureLevel.NORMAL

        if pressure != self.current_pressure:
            await self._handle_pressure_change(pressure, usage)

        self.current_pressure = pressure
        return pressure

    async def _handle_pressure_change(
        self, new_pressure: ResourcePressureLevel, usage: dict[str, Any]
    ) -> None:
        """Handle resource pressure level changes."""
        degradation_applied = []

        for strategy in self.degradation_strategies:
            if strategy.should_apply(new_pressure):
                degradation_applied.append(strategy.name)
                strategy.applied_count += 1

        circuit_breakers_open = [
            executor_id
            for executor_id, breaker in self.circuit_breakers.items()
            if breaker.state == CircuitState.OPEN
        ]

        event = ResourcePressureEvent(
            timestamp=datetime.utcnow(),
            pressure_level=new_pressure,
            cpu_percent=usage.get("cpu_percent", 0.0),
            memory_mb=usage.get("rss_mb", 0.0),
            memory_percent=usage.get("memory_percent", 0.0),
            worker_count=int(usage.get("worker_budget", 0)),
            active_executors=len(self._active_executors),
            degradation_applied=degradation_applied,
            circuit_breakers_open=circuit_breakers_open,
            message=f"Resource pressure changed: {self.current_pressure.value} -> {new_pressure.value}",
        )

        self.pressure_history.append(event)

        logger.warning(
            f"Resource pressure: {new_pressure.value}",
            extra={
                "cpu_percent": event.cpu_percent,
                "memory_mb": event.memory_mb,
                "memory_percent": event.memory_percent,
                "degradation_applied": degradation_applied,
                "circuit_breakers_open": circuit_breakers_open,
            },
        )

        if self.alert_callback:
            try:
                self.alert_callback(event)
            except Exception as exc:
                logger.error(f"Alert callback failed: {exc}")

    def get_degradation_config(
        self, executor_id: str
    ) -> dict[str, Any]:
        """Get degradation configuration for executor at current pressure."""
        config: dict[str, Any] = {
            "entity_limit_factor": 1.0,
            "disable_expensive_computations": False,
            "use_simplified_methods": False,
            "skip_optional_analysis": False,
            "reduce_embedding_dims": False,
            "applied_strategies": [],
        }

        if not self.enable_degradation:
            return config

        for strategy in self.degradation_strategies:
            if strategy.should_apply(self.current_pressure):
                config["entity_limit_factor"] = min(
                    config["entity_limit_factor"], strategy.entity_limit_factor
                )
                config["disable_expensive_computations"] = (
                    config["disable_expensive_computations"]
                    or strategy.disable_expensive_computations
                )
                config["use_simplified_methods"] = (
                    config["use_simplified_methods"] or strategy.use_simplified_methods
                )
                config["skip_optional_analysis"] = (
                    config["skip_optional_analysis"] or strategy.skip_optional_analysis
                )
                config["reduce_embedding_dims"] = (
                    config["reduce_embedding_dims"] or strategy.reduce_embedding_dims
                )
                config["applied_strategies"].append(strategy.name)

        return config

    async def allocate_resources(
        self, executor_id: str
    ) -> dict[str, Any]:
        """Allocate resources for executor based on priority and availability."""
        await self.assess_resource_pressure()

        policy = self.allocation_policies.get(
            executor_id,
            ResourceAllocationPolicy(
                executor_id=executor_id,
                priority=ExecutorPriority.NORMAL,
                min_memory_mb=64.0,
                max_memory_mb=256.0,
                min_workers=1,
                max_workers=4,
            ),
        )
        degradation = self.get_degradation_config(executor_id)

        max_memory = policy.max_memory_mb * degradation["entity_limit_factor"]
        max_workers = min(
            policy.max_workers,
            max(policy.min_workers, self.resource_limits.max_workers),
        )

        if self.current_pressure in [
            ResourcePressureLevel.CRITICAL,
            ResourcePressureLevel.EMERGENCY,
        ]:
            if policy.priority == ExecutorPriority.CRITICAL:
                max_workers = policy.max_workers
            elif policy.priority == ExecutorPriority.HIGH:
                max_workers = max(policy.min_workers, policy.max_workers - 2)
            else:
                max_workers = policy.min_workers

        return {
            "max_memory_mb": max_memory,
            "max_workers": max_workers,
            "priority": policy.priority.value,
            "degradation": degradation,
        }

    async def start_executor_execution(
        self, executor_id: str
    ) -> dict[str, Any]:
        """Start tracking executor execution."""
        async with self._lock:
            self._active_executors.add(executor_id)

        allocation = await self.allocate_resources(executor_id)

        if executor_id not in self.executor_metrics:
            self.executor_metrics[executor_id] = ExecutorMetrics(
                executor_id=executor_id
            )

        return allocation

    async def end_executor_execution(
        self,
        executor_id: str,
        success: bool,
        duration_ms: float,
        memory_mb: float | None = None,
    ) -> None:
        """End tracking executor execution and update metrics."""
        async with self._lock:
            self._active_executors.discard(executor_id)

        metrics = self.executor_metrics.get(executor_id)
        if not metrics:
            return

        metrics.total_executions += 1
        metrics.last_execution_time = datetime.utcnow()

        if success:
            metrics.successful_executions += 1
            if self.enable_circuit_breakers:
                breaker = self.get_or_create_circuit_breaker(executor_id)
                breaker.record_success()
        else:
            metrics.failed_executions += 1
            if self.enable_circuit_breakers:
                breaker = self.get_or_create_circuit_breaker(executor_id)
                breaker.record_failure(memory_mb)

        if memory_mb is not None:
            metrics.memory_samples.append(memory_mb)
            if len(metrics.memory_samples) > 100:
                metrics.memory_samples.pop(0)

            metrics.avg_memory_mb = sum(metrics.memory_samples) / len(
                metrics.memory_samples
            )
            metrics.peak_memory_mb = max(
                metrics.peak_memory_mb, memory_mb
            )

        metrics.duration_samples.append(duration_ms)
        if len(metrics.duration_samples) > 100:
            metrics.duration_samples.pop(0)

        metrics.avg_duration_ms = sum(metrics.duration_samples) / len(
            metrics.duration_samples
        )

    def get_executor_metrics(self, executor_id: str) -> dict[str, Any]:
        """Get metrics for specific executor."""
        metrics = self.executor_metrics.get(executor_id)
        if not metrics:
            return {}

        success_rate = 0.0
        if metrics.total_executions > 0:
            success_rate = (
                metrics.successful_executions / metrics.total_executions
            ) * 100

        breaker = self.circuit_breakers.get(executor_id)

        return {
            "executor_id": executor_id,
            "total_executions": metrics.total_executions,
            "successful_executions": metrics.successful_executions,
            "failed_executions": metrics.failed_executions,
            "success_rate_percent": success_rate,
            "avg_memory_mb": metrics.avg_memory_mb,
            "peak_memory_mb": metrics.peak_memory_mb,
            "avg_duration_ms": metrics.avg_duration_ms,
            "last_execution": (
                metrics.last_execution_time.isoformat()
                if metrics.last_execution_time
                else None
            ),
            "circuit_breaker_state": breaker.state.value if breaker else "closed",
        }

    def get_resource_status(self) -> dict[str, Any]:
        """Get comprehensive resource management status."""
        usage = self.resource_limits.get_resource_usage()

        executor_stats = {
            executor_id: self.get_executor_metrics(executor_id)
            for executor_id in self.executor_metrics
        }

        active_strategies = [
            {
                "name": strategy.name,
                "threshold": strategy.pressure_threshold.value,
                "applied_count": strategy.applied_count,
                "config": {
                    "entity_limit_factor": strategy.entity_limit_factor,
                    "disable_expensive_computations": strategy.disable_expensive_computations,
                    "use_simplified_methods": strategy.use_simplified_methods,
                    "skip_optional_analysis": strategy.skip_optional_analysis,
                    "reduce_embedding_dims": strategy.reduce_embedding_dims,
                },
            }
            for strategy in self.degradation_strategies
            if strategy.should_apply(self.current_pressure)
        ]

        circuit_breaker_summary = {
            executor_id: {
                "state": breaker.state.value,
                "failure_count": breaker.failure_count,
                "last_failure": (
                    breaker.last_failure_time.isoformat()
                    if breaker.last_failure_time
                    else None
                ),
            }
            for executor_id, breaker in self.circuit_breakers.items()
        }

        recent_pressure = list(self.pressure_history)[-10:]

        return {
            "timestamp": datetime.utcnow().isoformat(),
            "current_pressure": self.current_pressure.value,
            "resource_usage": usage,
            "active_executors": list(self._active_executors),
            "executor_metrics": executor_stats,
            "active_degradation_strategies": active_strategies,
            "circuit_breakers": circuit_breaker_summary,
            "recent_pressure_events": [
                {
                    "timestamp": event.timestamp.isoformat(),
                    "level": event.pressure_level.value,
                    "cpu_percent": event.cpu_percent,
                    "memory_mb": event.memory_mb,
                    "message": event.message,
                }
                for event in recent_pressure
            ],
        }

    def register_allocation_policy(
        self, policy: ResourceAllocationPolicy
    ) -> None:
        """Register custom resource allocation policy for executor."""
        self.allocation_policies[policy.executor_id] = policy
        logger.info(
            f"Registered allocation policy for {policy.executor_id}: "
            f"priority={policy.priority.value}"
        )

    def reset_circuit_breaker(self, executor_id: str) -> bool:
        """Manually reset circuit breaker for executor."""
        breaker = self.circuit_breakers.get(executor_id)
        if not breaker:
            return False

        breaker.state = CircuitState.CLOSED
        breaker.failure_count = 0
        breaker.success_count = 0
        breaker.last_state_change = datetime.utcnow()

        logger.info(f"Circuit breaker reset for {executor_id}")
        return True



################################################################################
# FILE: phase2_m_signature_runtime_validator.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_m_signature_runtime_validator.py
################################################################################

"""
Module: phase2_m_signature_runtime_validator
PHASE_LABEL: Phase 2
Sequence: M

"""
"""
Runtime Signature Validation for Chain Layer

Provides runtime validation of method calls against signature definitions:
- Validates required inputs are present (hard failure if missing)
- Warns about missing critical optional inputs (penalty but no hard failure)
- Tracks optional input usage
- Validates output types and ranges

This module integrates with the orchestrator to ensure method calls comply
with chain layer signature contracts.
"""

import logging
from pathlib import Path
from typing import Any, TypedDict

from orchestration.method_signature_validator import (
    MethodSignatureValidator,
)

logger = logging.getLogger(__name__)


class ValidationResult(TypedDict):
    passed: bool
    hard_failures: list[str]
    soft_failures: list[str]
    warnings: list[str]
    missing_critical_optional: list[str]


class SignatureRuntimeValidator:
    """
    Runtime validator for method signatures in the analysis chain.

    Enforces signature contracts at runtime:
    - Required inputs: MUST be present, raises exception if missing
    - Critical optional: Should be present, logs warning and applies penalty
    - Optional inputs: Nice to have, no penalty
    """

    def __init__(
        self,
        signatures_path: Path | str | None = None,
        strict_mode: bool = True,
        penalty_for_missing_critical: float = 0.1,
    ) -> None:
        if signatures_path is None:
            signatures_path = Path(
                "config/json_files_ no_schemas/method_signatures.json"
            )

        self.validator = MethodSignatureValidator(signatures_path)
        self.validator.load_signatures()
        self.strict_mode = strict_mode
        self.penalty_for_missing_critical = penalty_for_missing_critical
        self._validation_stats: dict[str, dict[str, int]] = {}

    def validate_inputs(
        self, method_id: str, provided_inputs: dict[str, Any]
    ) -> ValidationResult:
        """
        Validate that provided inputs match method signature requirements.

        Args:
            method_id: Identifier for the method
            provided_inputs: Dictionary of input parameters provided to method

        Returns:
            ValidationResult with passed status and any failures/warnings
        """
        hard_failures = []
        soft_failures = []
        warnings = []
        missing_critical_optional = []

        # Get method signature
        signature = self.validator.get_method_signature(method_id)
        if signature is None:
            if self.strict_mode:
                hard_failures.append(f"Method signature not found for: {method_id}")
            else:
                warnings.append(f"Method signature not found for: {method_id}")

            return ValidationResult(
                passed=len(hard_failures) == 0,
                hard_failures=hard_failures,
                soft_failures=soft_failures,
                warnings=warnings,
                missing_critical_optional=missing_critical_optional,
            )

        # Check required inputs
        required_inputs = signature.get("required_inputs", [])
        for required_input in required_inputs:
            if required_input not in provided_inputs:
                hard_failures.append(
                    f"Required input '{required_input}' missing for method {method_id}"
                )
            elif provided_inputs[required_input] is None:
                hard_failures.append(
                    f"Required input '{required_input}' is None for method {method_id}"
                )

        # Check critical optional inputs
        critical_optional = signature.get("critical_optional", [])
        for critical_input in critical_optional:
            if (
                critical_input not in provided_inputs
                or provided_inputs[critical_input] is None
            ):
                missing_critical_optional.append(critical_input)
                soft_failures.append(
                    f"Critical optional input '{critical_input}' missing for method {method_id} "
                    f"(penalty: {self.penalty_for_missing_critical})"
                )

        # Track optional inputs usage (for statistics)
        optional_inputs = signature.get("optional_inputs", [])
        provided_optional = [
            inp
            for inp in optional_inputs
            if inp in provided_inputs and provided_inputs[inp] is not None
        ]

        if len(provided_optional) < len(optional_inputs):
            missing_optional = set(optional_inputs) - set(provided_optional)
            warnings.append(f"Optional inputs not provided: {missing_optional}")

        # Record validation stats
        if method_id not in self._validation_stats:
            self._validation_stats[method_id] = {
                "calls": 0,
                "hard_failures": 0,
                "soft_failures": 0,
            }

        self._validation_stats[method_id]["calls"] += 1
        if hard_failures:
            self._validation_stats[method_id]["hard_failures"] += 1
        if soft_failures:
            self._validation_stats[method_id]["soft_failures"] += 1

        passed = len(hard_failures) == 0

        return ValidationResult(
            passed=passed,
            hard_failures=hard_failures,
            soft_failures=soft_failures,
            warnings=warnings,
            missing_critical_optional=missing_critical_optional,
        )

    def validate_output(self, method_id: str, output: Any) -> ValidationResult:
        """
        Validate that method output matches signature specification.

        Args:
            method_id: Identifier for the method
            output: Output value from method execution

        Returns:
            ValidationResult with passed status and any failures/warnings
        """
        hard_failures = []
        soft_failures = []
        warnings = []

        signature = self.validator.get_method_signature(method_id)
        if signature is None:
            warnings.append(
                f"Method signature not found for output validation: {method_id}"
            )
            return ValidationResult(
                passed=True,
                hard_failures=[],
                soft_failures=[],
                warnings=warnings,
                missing_critical_optional=[],
            )

        # Validate output type
        expected_type = signature.get("output_type", "Any")
        if expected_type != "Any":
            actual_type = type(output).__name__
            if actual_type != expected_type:
                # Try some common conversions
                type_map = {
                    "float": (float, int),
                    "int": (int,),
                    "str": (str,),
                    "list": (list, tuple),
                    "dict": (dict,),
                    "bool": (bool,),
                }

                if expected_type in type_map:
                    if not isinstance(output, type_map[expected_type]):
                        soft_failures.append(
                            f"Output type mismatch for {method_id}: "
                            f"expected {expected_type}, got {actual_type}"
                        )
                else:
                    warnings.append(
                        f"Cannot validate output type for {method_id}: "
                        f"expected {expected_type}, got {actual_type}"
                    )

        # Validate output range
        output_range = signature.get("output_range")
        if output_range is not None and isinstance(output, (int, float)):
            min_val, max_val = output_range
            if not (min_val <= output <= max_val):
                soft_failures.append(
                    f"Output value {output} out of range [{min_val}, {max_val}] "
                    f"for method {method_id}"
                )

        passed = len(hard_failures) == 0

        return ValidationResult(
            passed=passed,
            hard_failures=hard_failures,
            soft_failures=soft_failures,
            warnings=warnings,
            missing_critical_optional=[],
        )

    def calculate_penalty(self, validation_result: ValidationResult) -> float:
        """
        Calculate penalty score based on validation failures.

        Args:
            validation_result: Result from validate_inputs

        Returns:
            Penalty value (0.0 = no penalty, higher = more severe)
        """
        penalty = 0.0

        # Hard failures result in maximum penalty
        if validation_result["hard_failures"]:
            return 1.0

        # Apply penalty for missing critical optional inputs
        num_missing_critical = len(validation_result["missing_critical_optional"])
        penalty += num_missing_critical * self.penalty_for_missing_critical

        # Soft failures add smaller penalty
        penalty += len(validation_result["soft_failures"]) * 0.05

        return min(penalty, 1.0)  # Cap at 1.0

    def get_validation_stats(self) -> dict[str, dict[str, int]]:
        """Get validation statistics for all methods."""
        return self._validation_stats.copy()

    def validate_method_call(
        self,
        method_id: str,
        provided_inputs: dict[str, Any],
        raise_on_failure: bool = True,
    ) -> tuple[bool, float, list[str]]:
        """
        Convenience method to validate a method call.

        Args:
            method_id: Method identifier
            provided_inputs: Input parameters
            raise_on_failure: Whether to raise exception on hard failures

        Returns:
            Tuple of (passed, penalty, messages)

        Raises:
            ValueError: If validation fails and raise_on_failure is True
        """
        result = self.validate_inputs(method_id, provided_inputs)
        penalty = self.calculate_penalty(result)

        messages = []
        if result["hard_failures"]:
            messages.extend(result["hard_failures"])
            if raise_on_failure:
                raise ValueError(
                    f"Method call validation failed for {method_id}:\n"
                    + "\n".join(result["hard_failures"])
                )

        if result["soft_failures"]:
            messages.extend(result["soft_failures"])
            for msg in result["soft_failures"]:
                logger.warning(msg)

        if result["warnings"]:
            messages.extend(result["warnings"])
            for msg in result["warnings"]:
                logger.debug(msg)

        return result["passed"], penalty, messages


# Global validator instance
_runtime_validator: SignatureRuntimeValidator | None = None


def get_runtime_validator(
    signatures_path: Path | str | None = None, strict_mode: bool = True
) -> SignatureRuntimeValidator:
    """Get or create global runtime validator instance."""
    global _runtime_validator

    if _runtime_validator is None:
        _runtime_validator = SignatureRuntimeValidator(
            signatures_path=signatures_path, strict_mode=strict_mode
        )

    return _runtime_validator


def validate_method_call(
    method_id: str, provided_inputs: dict[str, Any], raise_on_failure: bool = True
) -> tuple[bool, float, list[str]]:
    """
    Convenience function to validate a method call using global validator.

    Args:
        method_id: Method identifier
        provided_inputs: Input parameters
        raise_on_failure: Whether to raise exception on hard failures

    Returns:
        Tuple of (passed, penalty, messages)
    """
    validator = get_runtime_validator()
    return validator.validate_method_call(method_id, provided_inputs, raise_on_failure)



################################################################################
# FILE: phase2_n_task_planner.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_n_task_planner.py
################################################################################

"""
Module: phase2_n_task_planner
PHASE_LABEL: Phase 2
Sequence: N

"""
from __future__ import annotations

import logging
from dataclasses import dataclass
```

Phase 2 Source Code

```python
from datetime import datetime, timezone
from types import MappingProxyType
from typing import TYPE_CHECKING, Any, Protocol

if TYPE_CHECKING:
    from canonic_phases.Phase_two.irrigation_synchronizer import ChunkRoutingResult

logger = logging.getLogger(__name__)

EXPECTED_TASKS_PER_CHUNK = 5
EXPECTED_TASKS_PER_POLICY_AREA = 30
MAX_QUESTION_GLOBAL = 999


class RoutingResult(Protocol):
    """Protocol for routing result objects that provide policy_area_id."""

    policy_area_id: str


def _freeze_immutable(obj: Any) -> Any:  # noqa: ANN401
    if isinstance(obj, dict):
        return MappingProxyType({k: _freeze_immutable(v) for k, v in obj.items()})
    if isinstance(obj, list | tuple):
        return tuple(_freeze_immutable(x) for x in obj)
    if isinstance(obj, set):
        return frozenset(_freeze_immutable(x) for x in obj)
    return obj


@dataclass(frozen=True, slots=True)
class MicroQuestionContext:
    task_id: str
    question_id: str
    question_global: int
    policy_area_id: str
    dimension_id: str
    chunk_id: str
    base_slot: str
    cluster_id: str
    patterns: tuple[Any, ...]
    signals: Any
    expected_elements: tuple[Any, ...]
    signal_requirements: Any
    creation_timestamp: str

    def __post_init__(self) -> None:
        object.__setattr__(self, "patterns", tuple(self.patterns))
        object.__setattr__(self, "signals", _freeze_immutable(self.signals))
        object.__setattr__(self, "expected_elements", tuple(self.expected_elements))
        object.__setattr__(
            self, "signal_requirements", _freeze_immutable(self.signal_requirements)
        )


@dataclass(frozen=True, slots=True)
class ExecutableTask:
    task_id: str
    question_id: str
    question_global: int
    policy_area_id: str
    dimension_id: str
    chunk_id: str
    patterns: list[dict[str, Any]]
    signals: dict[str, Any]
    creation_timestamp: str
    expected_elements: list[dict[str, Any]]
    metadata: dict[str, Any]

    def __post_init__(self) -> None:
        if not self.task_id:
            raise ValueError("task_id cannot be empty")
        if not self.question_id:
            raise ValueError("question_id cannot be empty")
        if not isinstance(self.question_global, int):
            raise ValueError(
                f"question_global must be an integer, got {type(self.question_global).__name__}"
            )
        if not (0 <= self.question_global <= MAX_QUESTION_GLOBAL):
            raise ValueError(
                f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got {self.question_global}"
            )
        if not self.policy_area_id:
            raise ValueError("policy_area_id cannot be empty")
        if not self.dimension_id:
            raise ValueError("dimension_id cannot be empty")
        if not self.chunk_id:
            raise ValueError("chunk_id cannot be empty")
        if not self.creation_timestamp:
            raise ValueError("creation_timestamp cannot be empty")


def _validate_element_compatibility(  # noqa: PLR0912
    provisional_task_id: str,
    question_schema: list[dict[str, Any]] | dict[str, Any],
    chunk_schema: list[dict[str, Any]] | dict[str, Any],
    common_type_class: type,  # noqa: ARG001
) -> int:
    validated_count = 0

    if isinstance(question_schema, list) and isinstance(chunk_schema, list):
        for idx, (q_elem, c_elem) in enumerate(
            zip(question_schema, chunk_schema, strict=True)
        ):
            if q_elem.get("type") is None:
                raise ValueError(
                    f"Task {provisional_task_id}: Question element at index {idx} "
                    f"has missing type field"
                )
            if c_elem.get("type") is None:
                raise ValueError(
                    f"Task {provisional_task_id}: Chunk element at index {idx} "
                    f"has missing type field"
                )

            if q_elem["type"] != c_elem["type"]:
                raise ValueError(
                    f"Task {provisional_task_id}: Type mismatch at index {idx}: "
                    f"question type '{q_elem['type']}' != chunk type '{c_elem['type']}'"
                )

            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field mismatch at index {idx}: "
                    f"question requires element but chunk marks it optional"
                )

            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)
            if c_minimum < q_minimum:
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold mismatch at index {idx}: "
                    f"chunk minimum ({c_minimum}) is lower than question minimum ({q_minimum})"
                )

            validated_count += 1

    elif isinstance(question_schema, dict) and isinstance(chunk_schema, dict):
        sorted_keys = sorted(set(question_schema.keys()) & set(chunk_schema.keys()))
        for key in sorted_keys:
            q_elem = question_schema[key]
            c_elem = chunk_schema[key]

            if q_elem.get("type") is None:
                raise ValueError(
                    f"Task {provisional_task_id}: Question element '{key}' "
                    f"has missing type field"
                )
            if c_elem.get("type") is None:
                raise ValueError(
                    f"Task {provisional_task_id}: Chunk element '{key}' "
                    f"has missing type field"
                )

            if q_elem["type"] != c_elem["type"]:
                raise ValueError(
                    f"Task {provisional_task_id}: Type mismatch for key '{key}': "
                    f"question type '{q_elem['type']}' != chunk type '{c_elem['type']}'"
                )

            q_required = q_elem.get("required", False)
            c_required = c_elem.get("required", False)
            if q_required and not c_required:
                raise ValueError(
                    f"Task {provisional_task_id}: Required field mismatch for key '{key}': "
                    f"question requires element but chunk marks it optional"
                )

            q_minimum = q_elem.get("minimum", 0)
            c_minimum = c_elem.get("minimum", 0)
            if c_minimum < q_minimum:
                raise ValueError(
                    f"Task {provisional_task_id}: Threshold mismatch for key '{key}': "
                    f"chunk minimum ({c_minimum}) is lower than question minimum ({q_minimum})"
                )

            validated_count += 1

    return validated_count


def _validate_schema(question: dict[str, Any], chunk: dict[str, Any]) -> None:
    """Validate schema compatibility between question and chunk expected_elements.

    Performs shallow equality check and validates semantic constraints:
    - Asymmetric required field implication: if question element is required,
      chunk element must also be required
    - Minimum threshold ordering: chunk minimum must be >= question minimum

    Args:
        question: Question dict with expected_elements field
        chunk: Chunk dict with expected_elements field

    Raises:
        ValueError: If schema mismatch, required field implication violation,
                   or minimum threshold ordering violation detected
    """
    question_id = question.get("question_id", "UNKNOWN")
    q_elements = question.get("expected_elements", [])
    c_elements = chunk.get("expected_elements", [])

    if q_elements != c_elements:
        raise ValueError(
            f"Schema mismatch for question {question_id}: "
            f"expected_elements differ between question and chunk.\n"
            f"Question schema: {q_elements}\n"
            f"Chunk schema: {c_elements}"
        )

    if not isinstance(q_elements, list) or not isinstance(c_elements, list):
        return

    if len(q_elements) != len(c_elements):
        return

    for idx, (q_elem, c_elem) in enumerate(zip(q_elements, c_elements, strict=True)):
        if not isinstance(q_elem, dict) or not isinstance(c_elem, dict):
            continue

        q_required = q_elem.get("required", False)
        c_required = c_elem.get("required", False)

        if q_required and not c_required:
            element_type = q_elem.get("type", f"element_at_index_{idx}")
            raise ValueError(
                f"Required-field implication violation for question {question_id}: "
                f"element type '{element_type}' at index {idx} is required in question "
                f"but marked as optional in chunk"
            )

        q_minimum = q_elem.get("minimum", 0)
        c_minimum = c_elem.get("minimum", 0)

        if isinstance(q_minimum, (int, float)) and isinstance(c_minimum, (int, float)):
            if c_minimum < q_minimum:
                element_type = q_elem.get("type", f"element_at_index_{idx}")
                raise ValueError(
                    f"Minimum threshold ordering violation for question {question_id}: "
                    f"element type '{element_type}' at index {idx} has "
                    f"chunk minimum ({c_minimum}) < question minimum ({q_minimum})"
                )


def _construct_task(
    question: dict[str, Any],
    routing_result: ChunkRoutingResult,
    applicable_patterns: tuple[Any, ...],
    resolved_signals: tuple[Any, ...],
    generated_task_ids: set[str],
    correlation_id: str,
) -> ExecutableTask:
    question_id = question.get("question_id", "UNKNOWN")
    question_global = question.get("question_global")

    if question_global is None:
        raise ValueError(
            f"Task construction failure for {question_id}: "
            "question_global field missing or None"
        )

    if not isinstance(question_global, int):
        raise ValueError(
            f"Task construction failure for {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"Task construction failure for {question_id}: "
            f"question_global must be in range 0-{MAX_QUESTION_GLOBAL}, got {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)

    patterns_list = (
        list(applicable_patterns)
        if not isinstance(applicable_patterns, list)
        else applicable_patterns
    )

    signals_dict = {}
```

Phase 2 Source Code

```
    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

    expected_elements = question.get("expected_elements", [])
    expected_elements_list = (
        list(expected_elements) if isinstance(expected_elements, list | tuple) else []
    )

    document_position = routing_result.document_position

    metadata = {
        "base_slot": question.get("base_slot", ""),
        "cluster_id": question.get("cluster_id", ""),
        "document_position": document_position,
        "synchronizer_version": "2.0.0",
        "correlation_id": correlation_id,
        "original_pattern_count": len(applicable_patterns),
        "original_signal_count": len(resolved_signals),
        "filtered_pattern_count": len(patterns_list),
        "resolved_signal_count": len(signals_dict),
        "schema_element_count": len(expected_elements_list),
    }

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

    dimension_id = (
        routing_result.dimension_id
        if routing_result.dimension_id
        else question.get("dimension_id", "")
    )

    try:
        task = ExecutableTask(
            task_id=task_id,
            question_id=question.get("question_id", ""),
            question_global=question_global,
            policy_area_id=routing_result.policy_area_id,
            dimension_id=dimension_id,
            chunk_id=routing_result.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={routing_result.chunk_id}, pattern_count={len(patterns_list)}, "
        f"signal_count={len(signals_dict)}"
    )

    return task


def _construct_task_legacy(
    question: dict[str, Any],
    chunk: dict[str, Any],
    patterns: list[dict[str, Any]],
    signals: dict[str, Any],
    generated_task_ids: set[str],
    routing_result: RoutingResult,
) -> ExecutableTask:
    question_global = question.get("question_global")

    if not isinstance(question_global, int) or not (
        0 <= question_global <= MAX_QUESTION_GLOBAL
    ):
        raise ValueError(
            f"Invalid question_global: {question_global}. "
            f"Must be an integer in range 0-{MAX_QUESTION_GLOBAL}."
        )

    policy_area_id = routing_result.policy_area_id

    if question_global is None:
        raise ValueError("question_global is required")

    if not isinstance(question_global, int):
        raise ValueError(
            f"question_global must be an integer, got {type(question_global).__name__}"
        )

    if not (0 <= question_global <= MAX_QUESTION_GLOBAL):
        raise ValueError(
            f"question_global must be between 0 and {MAX_QUESTION_GLOBAL} inclusive, got {question_global}"
        )

    task_id = f"MQC-{question_global:03d}_{policy_area_id}"
    if task_id in generated_task_ids:
        question_id = question.get("question_id", "")
        raise ValueError(
            f"Duplicate task_id detected: {task_id} for question {question_id}"
        )

    generated_task_ids.add(task_id)

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

    expected_elements = question.get("expected_elements", [])
    expected_elements_list = (
        list(expected_elements) if isinstance(expected_elements, list | tuple) else []
    )
    patterns_list = list(patterns) if isinstance(patterns, list | tuple) else []

    signals_dict = dict(signals) if isinstance(signals, dict) else {}

    metadata = {
        "base_slot": question.get("base_slot", ""),
        "cluster_id": question.get("cluster_id", ""),
        "document_position": None,
        "synchronizer_version": "2.0.0",
        "correlation_id": "",
        "original_pattern_count": len(patterns_list),
        "original_signal_count": len(signals_dict),
        "filtered_pattern_count": len(patterns_list),
        "resolved_signal_count": len(signals_dict),
        "schema_element_count": len(expected_elements_list),
    }

    try:
        task = ExecutableTask(
            task_id=task_id,
            question_id=question.get("question_id", ""),
            question_global=question_global,
            policy_area_id=policy_area_id,
            dimension_id=question.get("dimension_id", ""),
            chunk_id=chunk.get("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

    return task



################################################################################
# FILE: phase2_o_methods_registry.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_o_methods_registry.py
################################################################################

"""
Module: phase2_o_methods_registry
PHASE_LABEL: Phase 2
Sequence: O
Description: Methods registry with lazy loading

Version: 1.0.0
Last Modified: 2025-12-20
Author: F.A.R.F.A.N Policy Pipeline
License: Proprietary

This module is part of Phase 2: Analysis & Question Execution.
All files in Phase_two/ must contain PHASE_LABEL: Phase 2.
"""
from __future__ import annotations

import logging
import threading
import time
import weakref
from dataclasses import dataclass
from importlib import import_module
from typing import Any, Callable

logger = logging.getLogger(__name__)


class MethodRegistryError(RuntimeError):
    """Raised when a method cannot be retrieved."""


@dataclass
class CacheEntry:
    """Cache entry with TTL tracking."""

    instance: Any
    created_at: float
    last_accessed: float
    access_count: int = 0

    def is_expired(self, ttl_seconds: float) -> bool:
        """Check if entry has exceeded TTL."""
        if ttl_seconds <= 0:
            return False
        return (time.time() - self.last_accessed) > ttl_seconds

    def touch(self) -> None:
        """Update access timestamp and counter."""
        self.last_accessed = time.time()
        self.access_count += 1


class MethodRegistry:
    """Registry for lazy method injection without full class instantiation.

    Features memory management with TTL-based eviction and weakref support.
    """

    def __init__(
        self,
        class_paths: dict[str, str] | None = None,
        cache_ttl_seconds: float = 300.0,
        enable_weakref: bool = False,
        max_cache_size: int = 100,
    ) -> None:
        """Initialize the method registry.

        Args:
            class_paths: Optional mapping of class names to import paths.
                        If None, uses default paths from class_registry.
            cache_ttl_seconds: Time-to-live for cache entries in seconds.
                              Set to 0 to disable TTL-based eviction.
            enable_weakref: If True, use weak references for instances.
            max_cache_size: Maximum number of instances to cache.
        """
        # Import class paths from existing registry
        if class_paths is None:
            from orchestration.class_registry import get_class_paths
            class_paths = dict(get_class_paths())

        self._class_paths = class_paths
        self._cache: dict[str, CacheEntry] = {}
        self._weakref_cache: dict[str, weakref.ref[Any]] = {}
        self._direct_methods: dict[tuple[str, str], Callable[..., Any]] = {}
        self._failed_classes: set[str] = set()
        self._lock = threading.Lock()
        self._cache_ttl_seconds = cache_ttl_seconds
        self._enable_weakref = enable_weakref
        self._max_cache_size = max_cache_size

        # Special instantiation rules (from original MethodExecutor)
        self._special_instantiation: dict[str, Callable[[type], Any]] = {}

        # Metrics
        self._cache_hits = 0
        self._cache_misses = 0
        self._evictions = 0
        self._total_instantiations = 0

    def inject_method(
        self,
        class_name: str,
        method_name: str,
        method: Callable[..., Any],
    ) -> None:
        """Directly inject a method without needing a class.

        This allows bypassing class instantiation entirely.

        Args:
            class_name: Virtual class name for routing
            method_name: Method name
            method: Callable to inject
        """
        key = (class_name, method_name)
        self._direct_methods[key] = method
        logger.info(
            "method_injected_directly",
            class_name=class_name,
            method_name=method_name,
        )

    def register_instantiation_rule(
        self,
        class_name: str,
        instantiator: Callable[[type], Any],
    ) -> None:
        """Register special instantiation logic for a class.

        Args:
            class_name: Class name requiring special instantiation
            instantiator: Function that takes class type and returns instance
        """
        self._special_instantiation[class_name] = instantiator
        logger.debug(
            "instantiation_rule_registered",
            class_name=class_name,
        )

    def _load_class(self, class_name: str) -> type:
        """Load a class type from import path.

        Args:
            class_name: Name of class to load

        Returns:
            Class type

        Raises:
            MethodRegistryError: If class cannot be loaded
        """
        if class_name not in self._class_paths:
            raise MethodRegistryError(
                f"Class '{class_name}' not found in registry paths"
            )

        path = self._class_paths[class_name]
        module_name, _, attr_name = path.rpartition(".")

        if not module_name:
            raise MethodRegistryError(
                f"Invalid path for '{class_name}': {path}"
            )

        try:
            module = import_module(module_name)
            cls = getattr(module, attr_name)

            if not isinstance(cls, type):
                raise MethodRegistryError(
                    f"'{class_name}' is not a class: {type(cls).__name__}"
                )

            return cls

        except ImportError as exc:
            raise MethodRegistryError(
                f"Cannot import class '{class_name}' from {path}: {exc}"
            ) from exc
        except AttributeError as exc:
            raise MethodRegistryError(
                f"Class '{attr_name}' not found in module {module_name}: {exc}"
            ) from exc

    def _instantiate_class(self, class_name: str, cls: type) -> Any:
        """Instantiate a class using special rules or default constructor.

        Args:
            class_name: Name of class (for special rule lookup)
            cls: Class type to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If instantiation fails
        """
        # Use special instantiation rule if registered
        if class_name in self._special_instantiation:
            try:
                instantiator = self._special_instantiation[class_name]
                instance = instantiator(cls)
                logger.debug(
                    "class_instantiated_with_special_rule",
                    class_name=class_name,
                )
                return instance
            except Exception as exc:
                raise MethodRegistryError(
                    f"Special instantiation failed for '{class_name}': {exc}"
                ) from exc

        # Default instantiation (no-args constructor)
        try:
            instance = cls()
            logger.debug(
                "class_instantiated_default",
                class_name=class_name,
            )
            return instance
        except Exception as exc:
            raise MethodRegistryError(
                f"Default instantiation failed for '{class_name}': {exc}"
            ) from exc

    def _get_instance(self, class_name: str) -> Any:
        """Get or create instance of a class (lazy + cached).
        Args:
            class_name: Name of class to instantiate

        Returns:
            Instance of the class

        Raises:
            MethodRegistryError: If class cannot be instantiated
        """
        # Check if already failed
        if class_name in self._failed_classes:
            raise MethodRegistryError(
                f"Class '{class_name}' previously failed to instantiate"
            )

        # Use a lock to ensure thread-safe instantiation
        with self._lock:
            # Check weakref cache first
            if self._enable_weakref and class_name in self._weakref_cache:
                instance = self._weakref_cache[class_name]()
                if instance is not None:
                    self._cache_hits += 1
                    logger.debug(
                        "class_retrieved_from_weakref_cache",
                        class_name=class_name,
                    )
                    return instance
                else:
                    # Weakref was garbage collected
                    del self._weakref_cache[class_name]

            # Check regular cache and evict if expired
            if class_name in self._cache:
                entry = self._cache[class_name]
                if entry.is_expired(self._cache_ttl_seconds):
                    logger.info(
                        "cache_entry_expired",
                        class_name=class_name,
                        age_seconds=time.time() - entry.created_at,
                        access_count=entry.access_count,
                    )
                    del self._cache[class_name]
                    self._evictions += 1
                else:
                    entry.touch()
                    self._cache_hits += 1
                    return entry.instance

            # Cache miss - need to instantiate
            self._cache_misses += 1

            # Evict oldest entries if cache is full
            self._evict_if_full()

            # Load and instantiate class
            try:
                cls = self._load_class(class_name)
                instance = self._instantiate_class(class_name, cls)
                self._total_instantiations += 1

                # Store in appropriate cache
                if self._enable_weakref:
                    self._weakref_cache[class_name] = weakref.ref(instance)
                    logger.info(
                        "class_instantiated_weakref",
                        class_name=class_name,
                    )
                else:
                    entry = CacheEntry(
                        instance=instance,
                        created_at=time.time(),
                        last_accessed=time.time(),
                        access_count=1,
                    )
                    self._cache[class_name] = entry
                    logger.info(
                        "class_instantiated_cached",
                        class_name=class_name,
                    )

                return instance

            except MethodRegistryError:
                # Mark as failed to avoid repeated attempts
                self._failed_classes.add(class_name)
                raise

    def _evict_if_full(self) -> None:
        """Evict oldest cache entries if cache size exceeds maximum."""
        if len(self._cache) <= self._max_cache_size:
            return

        # Sort by last accessed time and evict oldest
        sorted_entries = sorted(
            self._cache.items(),
            key=lambda x: x[1].last_accessed,
        )

        evict_count = len(self._cache) - self._max_cache_size
        for class_name, entry in sorted_entries[:evict_count]:
            logger.info(
                "cache_entry_evicted_size_limit",
                class_name=class_name,
                age_seconds=time.time() - entry.created_at,
                access_count=entry.access_count,
            )
            del self._cache[class_name]
            self._evictions += 1

    def get_method(
        self,
        class_name: str,
        method_name: str,
    ) -> Callable[..., Any]:
        """Get method callable with lazy instantiation.

        This is the main entry point for retrieving methods.

        Args:
            class_name: Name of class containing the method
            method_name: Name of method to retrieve

        Returns:
            Callable method (bound or injected)

        Raises:
            MethodRegistryError: If method cannot be retrieved
        """
        # Check for directly injected method first
        key = (class_name, method_name)
        if key in self._direct_methods:
            logger.debug(
                "method_retrieved_direct",
                class_name=class_name,
                method_name=method_name,
            )
            return self._direct_methods[key]

        # Get instance (lazy) and retrieve method
        try:
            instance = self._get_instance(class_name)
            method = getattr(instance, method_name)

            if not callable(method):
                raise MethodRegistryError(
                    f"'{class_name}.{method_name}' is not callable"
                )

            logger.debug(
                "method_retrieved_from_instance",
                class_name=class_name,
                method_name=method_name,
            )
            return method

        except AttributeError as exc:
            raise MethodRegistryError(
                f"Method '{method_name}' not found on class '{class_name}'"
            ) from exc

    def has_method(self, class_name: str, method_name: str) -> bool:
        """Check if a method is available (without instantiating).

        Args:
            class_name: Name of class
            method_name: Name of method

        Returns:
            True if method exists (or is directly injected)
        """
        # Check direct injection
        key = (class_name, method_name)
        if key in self._direct_methods:
            return True

        # Check if class is known and not failed
        if class_name in self._failed_classes:
            return False

        if class_name not in self._class_paths:
            return False

        # If instance exists, check method
        if class_name in self._cache:
            instance = self._cache[class_name].instance
            return hasattr(instance, method_name)

        # Otherwise, assume it exists (lazy check)
        # Full validation happens on first get_method() call
        return True

    def clear_cache(self) -> dict[str, Any]:
        """Clear all cached instances.

        This should be called between pipeline runs to prevent memory bloat.

        Returns:
            Statistics about cleared cache entries.
        """
        with self._lock:
            cache_size = len(self._cache)
            weakref_size = len(self._weakref_cache)

            stats = {
                "entries_cleared": cache_size,
                "weakrefs_cleared": weakref_size,
                "total_hits": self._cache_hits,
                "total_misses": self._cache_misses,
                "total_evictions": self._evictions,
                "total_instantiations": self._total_instantiations,
            }

            # Clear caches
            self._cache.clear()
            self._weakref_cache.clear()

            logger.info(
                "cache_cleared",
                **stats,
            )

            return stats

    def evict_expired(self) -> int:
        """Manually evict expired entries.

        Returns:
            Number of entries evicted.
        """
        with self._lock:
            expired = []
            for class_name, entry in self._cache.items():
                if entry.is_expired(self._cache_ttl_seconds):
                    expired.append(class_name)

            for class_name in expired:
                entry = self._cache[class_name]
                logger.info(
                    "cache_entry_evicted_manual",
                    class_name=class_name,
                    age_seconds=time.time() - entry.created_at,
                    access_count=entry.access_count,
                )
                del self._cache[class_name]
                self._evictions += 1

            return len(expired)

    def get_stats(self) -> dict[str, Any]:
        """Get registry statistics.

        Returns:
            Dictionary with registry stats including cache performance metrics
        """
        with self._lock:
            cache_entries = []
            for class_name, entry in self._cache.items():
                cache_entries.append({
                    "class_name": class_name,
                    "age_seconds": time.time() - entry.created_at,
                    "last_accessed_seconds_ago": time.time() - entry.last_accessed,
                    "access_count": entry.access_count,
                })

            hit_rate = 0.0
            total_accesses = self._cache_hits + self._cache_misses
            if total_accesses > 0:
                hit_rate = self._cache_hits / total_accesses

            return {
                "total_classes_registered": len(self._class_paths),
                "cached_instances": len(self._cache),
                "weakref_instances": len(self._weakref_cache),
                "failed_classes": len(self._failed_classes),
                "direct_methods_injected": len(self._direct_methods),
                "cache_hits": self._cache_hits,
                "cache_misses": self._cache_misses,
                "cache_hit_rate": hit_rate,
                "evictions": self._evictions,
                "total_instantiations": self._total_instantiations,
                "cache_ttl_seconds": self._cache_ttl_seconds,
                "max_cache_size": self._max_cache_size,
                "enable_weakref": self._enable_weakref,
                "cache_entries": cache_entries,
                "failed_class_names": list(self._failed_classes),
            }
```

```
def setup_default_instantiation_rules(registry: MethodRegistry) -> None:
    """Setup default special instantiation rules.

    These rules replicate the logic from the original MethodExecutor
    for classes that need non-default instantiation.

    Args:
        registry: MethodRegistry to configure
    """
    # MunicipalOntology - shared instance pattern
    ontology_instance = None

    def instantiate_ontology(cls: type) -> Any:
        nonlocal ontology_instance
        if ontology_instance is None:
            ontology_instance = cls()
        return ontology_instance

    registry.register_instantiation_rule("MunicipalOntology", instantiate_ontology)

    # SemanticAnalyzer, PerformanceAnalyzer, TextMiningEngine - need ontology
    def instantiate_with_ontology(cls: type) -> Any:
        if ontology_instance is None:
            raise MethodRegistryError(
                f"Cannot instantiate {cls.__name__}: MunicipalOntology not available"
            )
        return cls(ontology_instance)

    for class_name in ["SemanticAnalyzer", "PerformanceAnalyzer", "TextMiningEngine"]:
        registry.register_instantiation_rule(class_name, instantiate_with_ontology)

    # PolicyTextProcessor - needs ProcessorConfig
    def instantiate_policy_processor(cls: type) -> Any:
        try:
            from farfan_pipeline.processing.policy_processor import ProcessorConfig
            return cls(ProcessorConfig())
        except ImportError as exc:
            raise MethodRegistryError(
                "Cannot instantiate PolicyTextProcessor: ProcessorConfig unavailable"
            ) from exc

    registry.register_instantiation_rule("PolicyTextProcessor", instantiate_policy_processor)


__all__ = [
    "MethodRegistry",
    "MethodRegistryError",
    "setup_default_instantiation_rules",
]



################################################################################
# FILE: phase2_p_executor_profiler.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_p_executor_profiler.py
################################################################################

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

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Executor Instrumentation and Profiling
PHASE_ROLE: Runtime performance measurement and regression detection for contract execution

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:
- 300 executor contracts 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 (supports both question_id and legacy base_slot)
    profiler = ExecutorProfiler()
    with profiler.profile_executor("Q001"):  # or legacy "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)

        # Initialize memory tracking (psutil)
        self._initialize_memory_tracking()

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

    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,
        }

    def _initialize_memory_tracking(self) -> None:
        """Initialize psutil for memory tracking if available and enabled.

        Sets self._psutil and self._psutil_process if psutil is available.
        Disables memory_tracking and emits structured warning if psutil unavailable.

        Postconditions:
            - If memory_tracking=True and psutil available: self._psutil_process is not None
            - If psutil unavailable: self.memory_tracking=False and warning logged
        """
        self._psutil = None
        self._psutil_process = None

        if not self.memory_tracking:
            return

        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

    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 executorâ\206\222dispensary 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 += """
```

Phase 2 Source Code

```
    <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",
]



################################################################################
# FILE: phase2_q_executor_instrumentation_mixin.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_q_executor_instrumentation_mixin.py
################################################################################

"""
Executor Instrumentation Mixin for Calibration Integration.

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Executor Instrumentation
PHASE_ROLE: Adds calibration instrumentation to executors for runtime metrics capture

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:
    # 300-contract model (preferred)
    class Q001_Executor(BaseExecutor, ExecutorInstrumentationMixin):
        def execute(self, context):
            # Instrumentation is automatically applied via wrapper
            ...
"""

from __future__ import annotations

import time
```

Phase 2 Source Code

```python
from typing import Any, Dict, Optional

from farfan_pipeline.phases.Phase_two.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"]



################################################################################
# FILE: phase2_r_executor_calibration_integration.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_r_executor_calibration_integration.py
################################################################################

"""Executor Calibration Integration - Stub Implementation.

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Executor Calibration Integration
PHASE_ROLE: Provides calibration instrumentation interface for executor performance tracking

This module provides the calibration instrumentation interface used by
ExecutorInstrumentationMixin. It captures runtime metrics and retrieves
quality scores from the calibration system.

NOTE: This is a stub implementation that satisfies the contract without
breaking imports. Full calibration integration is handled by the calibration_policy
module and the broader calibration system.

Design by Contract:
- Preconditions: executor_id is non-empty string, metrics are non-negative
- Postconditions: CalibrationResult always returned with valid score [0,1]
- Invariants: No side effects on external state, deterministic for same inputs
"""

from __future__ import annotations

import logging
from dataclasses import dataclass, field
from typing import Any

logger = logging.getLogger(__name__)


@dataclass
class CalibrationMetrics:
    """Runtime metrics captured during executor execution.

    Attributes:
        runtime_ms: Execution time in milliseconds
        memory_mb: Memory usage in megabytes
        methods_executed: Total number of methods called
        methods_succeeded: Number of methods that completed successfully
    """
    runtime_ms: float
    memory_mb: float
    methods_executed: int
    methods_succeeded: int


@dataclass
class CalibrationResult:
    """Result of calibration instrumentation with quality scores.

    Attributes:
        quality_score: Aggregated quality score [0,1]
        layer_scores: Per-layer quality scores
        layers_used: List of calibration layers applied
        aggregation_method: Method used to aggregate layer scores
        metrics: Runtime metrics captured during execution
    """
    quality_score: float
    layer_scores: dict[str, float] = field(default_factory=dict)
    layers_used: list[str] = field(default_factory=list)
    aggregation_method: str = "stub"
    metrics: CalibrationMetrics = field(default_factory=lambda: CalibrationMetrics(0.0, 0.0, 0, 0))


def instrument_executor(
    executor_id: str,
    context: dict[str, Any],
    runtime_ms: float,
    memory_mb: float,
    methods_executed: int,
    methods_succeeded: int,
) -> CalibrationResult:
    """Instrument executor execution with calibration data.

    This is a stub implementation that returns neutral calibration scores.
    Full calibration integration should be implemented when the calibration
    system is fully operational.

    Args:
        executor_id: Unique executor identifier
        context: Execution context
        runtime_ms: Execution time in milliseconds
        memory_mb: Memory usage in megabytes
        methods_executed: Total number of methods called
        methods_succeeded: Number of methods that completed successfully

    Returns:
        CalibrationResult with quality scores and metrics

    Preconditions:
        - executor_id is non-empty string
        - runtime_ms >= 0
        - memory_mb >= 0
        - methods_executed >= 0
        - methods_succeeded >= 0
        - methods_succeeded <= methods_executed

    Postconditions:
        - quality_score in [0, 1]
        - metrics match input values
    """
    if not executor_id:
        raise ValueError("executor_id cannot be empty")
    if runtime_ms < 0:
        raise ValueError(f"runtime_ms must be non-negative, got {runtime_ms}")
    if memory_mb < 0:
        raise ValueError(f"memory_mb must be non-negative, got {memory_mb}")
    if methods_executed < 0:
        raise ValueError(f"methods_executed must be non-negative, got {methods_executed}")
    if methods_succeeded < 0:
        raise ValueError(f"methods_succeeded must be non-negative, got {methods_succeeded}")
    if methods_succeeded > methods_executed:
        raise ValueError(
            f"methods_succeeded ({methods_succeeded}) cannot exceed "
            f"methods_executed ({methods_executed})"
        )

    metrics = CalibrationMetrics(
        runtime_ms=runtime_ms,
        memory_mb=memory_mb,
        methods_executed=methods_executed,
        methods_succeeded=methods_succeeded,
    )

    # Stub implementation: return neutral quality score
    # TODO: Integrate with full calibration system
    quality_score = 0.75  # Neutral baseline

    logger.debug(
        f"Calibration stub called for {executor_id}: "
        f"runtime={runtime_ms:.1f}ms, memory={memory_mb:.1f}MB, "
        f"methods={methods_executed}/{methods_succeeded}"
    )

    return CalibrationResult(
        quality_score=quality_score,
        layer_scores={},
        layers_used=[],
        aggregation_method="stub",
        metrics=metrics,
    )


def get_executor_config(
    executor_id: str,
    dimension: str,
    question: str,
) -> dict[str, Any]:
    """Get runtime configuration for executor.

    This is a stub implementation that returns conservative defaults.
    Full configuration loading should be implemented when the configuration
    system is fully operational.

    Args:
        executor_id: Unique executor identifier
        dimension: Dimension identifier (e.g., "D1")
        question: Question identifier (e.g., "Q1")

    Returns:
        Runtime configuration dictionary with HOW parameters

    Preconditions:
        - executor_id is non-empty string
        - dimension is non-empty string
        - question is non-empty string

    Postconditions:
        - Returns valid configuration dict
        - All required keys present with conservative defaults
    """
    if not executor_id:
        raise ValueError("executor_id cannot be empty")
    if not dimension:
        raise ValueError("dimension cannot be empty")
    if not question:
        raise ValueError("question cannot be empty")

    logger.debug(
        f"Config stub called for {executor_id} "
        f"(dimension={dimension}, question={question})"
    )

    # Stub implementation: return conservative defaults
    # TODO: Load from actual configuration files
    return {
        "timeout_seconds": 300,
        "max_retries": 3,
        "retry_delay_seconds": 1.0,
        "memory_limit_mb": 1024,
        "enable_caching": True,
        "enable_profiling": True,
    }


__all__ = [
    "CalibrationMetrics",
    "CalibrationResult",
    "instrument_executor",
    "get_executor_config",
]



################################################################################
# FILE: phase2_s_executor_config.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_s_executor_config.py
################################################################################

"""
ExecutorConfig: Runtime parametrization for executors (HOW we execute).

PHASE_LABEL: Phase 2

CRITICAL SEPARATION:
- This file contains ONLY runtime parameters (timeout, retry, etc.)
- NO calibration values (quality scores, fusion weights) are stored here
- Calibration data (WHAT quality) is loaded from:
  * src/cross_cutting_infrastructure/capaz_calibration_parmetrization/calibration/COHORT_2024_intrinsic_calibration.json
  * canonic_questionnaire_central/questionnaire_monolith.json

Loading hierarchy (highest to lowest priority):
1. CLI arguments (--timeout-s=120)
2. Environment variables (FARFAN_TIMEOUT_S=120)
3. Environment file (system/config/environments/{env}.json)
4. Executor config file (executor_configs/{executor_id}.json)
5. Conservative defaults

See CALIBRATION_VS_PARAMETRIZATION.md for complete specification.
"""

from __future__ import annotations

import json
```

Phase 2 Source Code

```
import os
from dataclasses import dataclass, field
from pathlib import Path
from typing import Any, Dict, Optional


@dataclass
class ExecutorConfig:
    """
    Runtime configuration for executor execution (HOW parameters only).

    This dataclass contains ONLY execution parameters that control HOW
    executors run, NOT calibration values that define WHAT quality we measure.

    Loading Hierarchy:
        CLI args > ENV vars > environment file > executor config file > defaults

    Attributes:
        timeout_s: Maximum execution time in seconds
        retry: Number of retry attempts on failure
        temperature: LLM sampling temperature (0.0 = deterministic)
        max_tokens: Maximum LLM output tokens
        memory_limit_mb: Memory limit in megabytes
        enable_profiling: Whether to enable execution profiling
        seed: Random seed for reproducibility
        extra: Additional executor-specific parameters
    """

    timeout_s: float | None = None
    retry: int | None = None
    temperature: float | None = None
    max_tokens: int | None = None
    memory_limit_mb: int | None = None
    enable_profiling: bool = True
    seed: int | None = None
    extra: dict[str, Any] | None = None

    def __post_init__(self) -> None:
        if self.timeout_s is not None and self.timeout_s <= 0:
            raise ValueError("timeout_s must be positive when provided")
        if self.max_tokens is not None and self.max_tokens <= 0:
            raise ValueError("max_tokens must be positive when provided")
        if self.retry is not None and self.retry < 0:
            raise ValueError("retry must be non-negative when provided")
        if self.temperature is not None and not (0.0 <= self.temperature <= 2.0):
            raise ValueError("temperature must be in range [0.0, 2.0]")
        if self.memory_limit_mb is not None and self.memory_limit_mb <= 0:
            raise ValueError("memory_limit_mb must be positive when provided")

    @classmethod
    def from_dict(cls, config_dict: Dict[str, Any]) -> ExecutorConfig:
        """Create ExecutorConfig from dictionary."""
        valid_fields = {
            "timeout_s", "retry", "temperature", "max_tokens",
            "memory_limit_mb", "enable_profiling", "seed", "extra"
        }
        filtered = {k: v for k, v in config_dict.items() if k in valid_fields}
        return cls(**filtered)

    @classmethod
    def load_from_sources(
        cls,
        executor_id: str,
        environment: str = "production",
        cli_overrides: Optional[Dict[str, Any]] = None
    ) -> ExecutorConfig:
        """
        Load ExecutorConfig from multiple sources with proper hierarchy.

        Loading order (highest to lowest priority):
        1. CLI arguments (passed via cli_overrides)
        2. Environment variables (FARFAN_*)
        3. Environment file (system/config/environments/{env}.json)
        4. Executor config file (executor_configs/{executor_id}.json)
        5. Conservative defaults

        Args:
            executor_id: Executor identifier (e.g., "Q001" or legacy "D3_Q2_TargetProportionalityAnalyzer")
            environment: Environment name (development, staging, production)
            cli_overrides: CLI argument overrides

        Returns:
            ExecutorConfig with merged configuration
        """
        config = cls._get_conservative_defaults()

        executor_config = cls._load_executor_config_file(executor_id)
        if executor_config:
            config.update(executor_config)

        env_config = cls._load_environment_file(environment)
        if env_config and "executor" in env_config:
            config.update(env_config["executor"])

        env_vars = cls._load_environment_variables()
        config.update(env_vars)
        if cli_overrides:
            config.update(cli_overrides)

        return cls.from_dict(config)

    @staticmethod
    def _get_conservative_defaults() -> Dict[str, Any]:
        """Get conservative default parameters."""
        return {
            "timeout_s": 300.0,
            "retry": 3,
            "temperature": 0.0,
            "max_tokens": 4096,
            "memory_limit_mb": 512,
            "enable_profiling": True,
            "seed": 42,
        }

    @staticmethod
    def _load_executor_config_file(executor_id: str) -> Optional[Dict[str, Any]]:
        """Load executor-specific config file."""
        config_file = Path(__file__).parent / "executor_configs" / f"{executor_id}.json"

        if not config_file.exists():
            return None

        try:
            with open(config_file) as f:
                data = json.load(f)
                return data.get("runtime_parameters", {})
        except (json.JSONDecodeError, IOError):
            return None

    @staticmethod
    def _load_environment_file(environment: str) -> Optional[Dict[str, Any]]:
        """Load environment-specific config file."""
        base_path = Path(__file__).parent.parent.parent.parent / "system" / "config" / "environments"
        env_file = base_path / f"{environment}.json"

        if not env_file.exists():
            return None

        try:
            with open(env_file) as f:
                return json.load(f)
        except (json.JSONDecodeError, IOError):
            return None

    @staticmethod
    def _load_environment_variables() -> Dict[str, Any]:
        """Load configuration from environment variables."""
        config = {}

        if "FARFAN_TIMEOUT_S" in os.environ:
            config["timeout_s"] = float(os.environ["FARFAN_TIMEOUT_S"])
        if "FARFAN_RETRY" in os.environ:
            config["retry"] = int(os.environ["FARFAN_RETRY"])
        if "FARFAN_TEMPERATURE" in os.environ:
            config["temperature"] = float(os.environ["FARFAN_TEMPERATURE"])
        if "FARFAN_MAX_TOKENS" in os.environ:
            config["max_tokens"] = int(os.environ["FARFAN_MAX_TOKENS"])
        if "FARFAN_MEMORY_LIMIT_MB" in os.environ:
            config["memory_limit_mb"] = int(os.environ["FARFAN_MEMORY_LIMIT_MB"])
        if "FARFAN_SEED" in os.environ:
            config["seed"] = int(os.environ["FARFAN_SEED"])

        return config

    def to_dict(self) -> Dict[str, Any]:
        """Convert to dictionary, excluding None values."""
        return {
            k: v for k, v in {
                "timeout_s": self.timeout_s,
                "retry": self.retry,
                "temperature": self.temperature,
                "max_tokens": self.max_tokens,
                "memory_limit_mb": self.memory_limit_mb,
                "enable_profiling": self.enable_profiling,
                "seed": self.seed,
                "extra": self.extra,
            }.items() if v is not None
        }


__all__ = ["ExecutorConfig"]



################################################################################
# FILE: phase2_t_irrigation_synchronizer.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_t_irrigation_synchronizer.py
################################################################################

"""Irrigation Synchronizer - Questionâ\206\222Chunkâ\206\222Taskâ\206\222Plan Coordination.
```

Phase 2 Source Code

```
PHASE_LABEL: Phase 2
This module implements the synchronization layer that maps questionnaire questions
to document chunks, generating an ExecutionPlan with 300 tasks (6 dimensions Ã\227 50
questions/dimension Ã\227 10 policy areas) for deterministic pipeline execution.

Architecture:
- IrrigationSynchronizer: Orchestrates chunkâ\206\222questionâ\206\222taskâ\206\222plan 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.schema_validation import (
    validate_phase6_schema_compatibility,
)
from farfan_pipeline.core.types import ChunkData, PreprocessedDocument
from canonic_phases.Phase_two.synchronization import ChunkMatrix

# Import executor-chunk synchronizer for JOIN table
try:
    from canonic_phases.Phase_two.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(
                "canonic_phases.Phase_two.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(
            "canonic_phases.Phase_two.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(
            "canonic_phases.Phase_two.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(
            "canonic_phases.Phase_two.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:
```

Phase 2 Source Code

```
    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)
```

Phase 2 Source Code

```
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
            )
```

Phase 2 Source Code

```
        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",
]



################################################################################
# FILE: phase2_u_synchronization.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_u_synchronization.py
################################################################################

"""
Module: phase2_u_synchronization
PHASE_LABEL: Phase 2
Sequence: U
Description: Synchronization utilities

Version: 1.0.0
Last Modified: 2025-12-20
Author: F.A.R.F.A.N Policy Pipeline
License: Proprietary

This module is part of Phase 2: Analysis & Question Execution.
All files in Phase_two/ must contain PHASE_LABEL: Phase 2.
"""
from __future__ import annotations

import hashlib
import json
import logging
import re
from dataclasses import dataclass
from typing import Any, Mapping, Sequence

logger = logging.getLogger(__name__)

_PA_RE = re.compile(r"^PA(0[1-9]|10)$")
_DIM_RE = re.compile(r"^DIM0[1-6]$")
_CHUNK_ID_RE = re.compile(r"^PA(0[1-9]|10)-DIM0[1-6]$")


def _get_mapping_value(obj: Any, key: str) -> Any:  # noqa: ANN401
    if isinstance(obj, Mapping):
        return obj.get(key)
    return None


def _get_attr_or_key(obj: Any, name: str) -> Any:  # noqa: ANN401
    value = _get_mapping_value(obj, name)
    if value is not None:
        return value
    return getattr(obj, name, None)


def _coerce_id(value: Any) -> str | None:  # noqa: ANN401
    if isinstance(value, str):
        return value
    if hasattr(value, "value") and isinstance(value.value, str):
        return value.value
    return None


def _extract_policy_area_id(chunk: Any) -> str:
    pa_id = _coerce_id(_get_attr_or_key(chunk, "policy_area_id"))
    if pa_id is None:
        pa_id = _coerce_id(_get_attr_or_key(chunk, "policy_area"))
    if pa_id is None:
        chunk_id = _coerce_id(_get_attr_or_key(chunk, "chunk_id")) or _coerce_id(
            _get_attr_or_key(chunk, "id")
        )
        if chunk_id:
            normalized = chunk_id.replace("_", "-")
            if _CHUNK_ID_RE.match(normalized):
                return normalized.split("-", 1)[0]
    if pa_id is None:
        raise ValueError("Chunk missing policy_area_id")
    return pa_id


def _extract_dimension_id(chunk: Any) -> str:
    dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension_id"))
    if dim_id is None:
        dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension"))
    if dim_id is None:
        dim_id = _coerce_id(_get_attr_or_key(chunk, "dimension_causal"))
    if dim_id is None:
        chunk_id = _coerce_id(_get_attr_or_key(chunk, "chunk_id")) or _coerce_id(
            _get_attr_or_key(chunk, "id")
        )
        if chunk_id:
            normalized = chunk_id.replace("_", "-")
            if _CHUNK_ID_RE.match(normalized):
                return normalized.split("-", 1)[1]
    if dim_id is None:
        raise ValueError("Chunk missing dimension_id")
    return dim_id


def _extract_text(chunk: Any) -> str:
    text = _get_attr_or_key(chunk, "text")
    if not isinstance(text, str):
        raise ValueError(f"Chunk text must be str, got {type(text).__name__}")
    return text


def _extract_document_position(chunk: Any) -> tuple[int, int] | None:
    start = _get_attr_or_key(chunk, "start_offset")
    end = _get_attr_or_key(chunk, "end_offset")
    if isinstance(start, int) and isinstance(end, int):
        if start < 0 or end < start:
            raise ValueError(f"Invalid document position: ({start}, {end})")
        return (start, end)

    start = _get_attr_or_key(chunk, "start_pos")
    end = _get_attr_or_key(chunk, "end_pos")
    if isinstance(start, int) and isinstance(end, int):
        if start < 0 or end < start:
            raise ValueError(f"Invalid document position: ({start}, {end})")
        return (start, end)

    text_span = _get_attr_or_key(chunk, "text_span")
    if text_span is not None:
        start = getattr(text_span, "start", None)
        end = getattr(text_span, "end", None)
        if isinstance(start, int) and isinstance(end, int):
            if start < 0 or end < start:
                raise ValueError(f"Invalid document position: ({start}, {end})")
            return (start, end)

    return None


def _validate_ids(policy_area_id: str, dimension_id: str) -> None:
    if not _PA_RE.match(policy_area_id):
        raise ValueError(f"Invalid policy_area_id: {policy_area_id}")
    if not _DIM_RE.match(dimension_id):
        raise ValueError(f"Invalid dimension_id: {dimension_id}")


def _validate_chunk_identity(
    chunk: Any,
    *,
    expected_chunk_id: str,
) -> None:
    if not _CHUNK_ID_RE.match(expected_chunk_id):
        raise ValueError(f"Invalid chunk_id format: {expected_chunk_id}")

    declared = _coerce_id(_get_attr_or_key(chunk, "chunk_id"))
    if declared is not None and declared != expected_chunk_id:
        raise ValueError(
            f"Chunk identity mismatch: expected chunk_id={expected_chunk_id} but got {declared}"
        )

    legacy_id = _coerce_id(_get_attr_or_key(chunk, "id"))
    if legacy_id is not None:
        normalized = legacy_id.replace("_", "-")
        if normalized != expected_chunk_id:
            raise ValueError(
                f"Chunk identity mismatch: expected chunk_id={expected_chunk_id} but got id={legacy_id}"
            )


@dataclass(frozen=True, slots=True)
class SmartPolicyChunk:
    chunk_id: str
    policy_area_id: str
    dimension_id: str
    text: str
    document_position: tuple[int, int] | None
    raw_chunk: Any | None = None

    @property
    def start_pos(self) -> int | None:
        if self.document_position is None:
            return None
        return self.document_position[0]

    @property
    def end_pos(self) -> int | None:
        if self.document_position is None:
            return None
        return self.document_position[1]


class ChunkMatrix:
    """60-slot PAÃ\227DIM chunk matrix with strict invariant validation."""

    EXPECTED_CHUNK_COUNT = 60

    def __init__(self, document: Any) -> None:  # noqa: ANN401
        self._preprocessed_document = document
        chunks = self._extract_chunks(document)
        self._chunk_matrix = self._build_matrix(chunks)
        self._matrix_keys_sorted = tuple(sorted(self._chunk_matrix.keys()))
        self._integrity_hash = self._compute_integrity_hash()

    @property
    def chunk_matrix(self) -> dict[tuple[str, str], SmartPolicyChunk]:
        return dict(self._chunk_matrix)

    @property
    def matrix_keys_sorted(self) -> tuple[tuple[str, str], ...]:
        return self._matrix_keys_sorted

    @property
    def integrity_hash(self) -> str:
        return self._integrity_hash

    def get_chunk(self, policy_area_id: str, dimension_id: str) -> SmartPolicyChunk:
        return self._chunk_matrix[(policy_area_id, dimension_id)]

    @staticmethod
    def _extract_chunks(document: Any) -> list[Any]:  # noqa: ANN401
        if document is None:
            raise ValueError("document is required")

        chunks = _get_attr_or_key(document, "chunks")
        if isinstance(chunks, Sequence) and not isinstance(chunks, (str, bytes)):
            return list(chunks)

        chunk_graph = _get_attr_or_key(document, "chunk_graph")
        if chunk_graph is not None:
            graph_chunks = _get_attr_or_key(chunk_graph, "chunks")
            if isinstance(graph_chunks, Mapping):
                return list(graph_chunks.values())

        if isinstance(document, Sequence) and not isinstance(document, (str, bytes)):
            return list(document)

        raise TypeError(
            "Unsupported document type for ChunkMatrix; expected .chunks sequence, "
            ".chunk_graph.chunks mapping, or a sequence of chunks"
        )

    @classmethod
    def _build_matrix(cls, chunks: Sequence[Any]) -> dict[tuple[str, str], SmartPolicyChunk]:
        chunk_matrix: dict[tuple[str, str], SmartPolicyChunk] = {}

        inserted_count = 0
        for chunk in chunks:
            inserted_count += 1
            policy_area_id = _extract_policy_area_id(chunk)
            dimension_id = _extract_dimension_id(chunk)
            _validate_ids(policy_area_id, dimension_id)

            expected_chunk_id = f"{policy_area_id}-{dimension_id}"
            _validate_chunk_identity(chunk, expected_chunk_id=expected_chunk_id)

            text = _extract_text(chunk)
            if not text.strip():
                raise ValueError(f"Chunk {expected_chunk_id} has empty text")
            document_position = _extract_document_position(chunk)

            key = (policy_area_id, dimension_id)
            if key in chunk_matrix:
                raise ValueError(
                    f"Duplicate chunk slot detected for key={key}. "
                    f"Inserted={inserted_count}, unique={len(chunk_matrix)}"
                )

            chunk_matrix[key] = SmartPolicyChunk(
                chunk_id=expected_chunk_id,
                policy_area_id=policy_area_id,
                dimension_id=dimension_id,
                text=text,
                document_position=document_position,
                raw_chunk=chunk,
            )

        if len(chunk_matrix) != cls.EXPECTED_CHUNK_COUNT:
            raise ValueError(
                "Chunk Matrix Invariant Violation: Expected 60 unique (PA, DIM) chunks "
                f"but found {len(chunk_matrix)}"
            )

        expected_keys = {
            (f"PA{pa:02d}", f"DIM{dim:02d}") for pa in range(1, 11) for dim in range(1, 7)
        }
        missing = expected_keys - set(chunk_matrix.keys())
        if missing:
            raise ValueError(f"Missing chunk combinations: {sorted(missing)}")

        chunks_per_policy_area = {
            f"PA{pa:02d}": sum(1 for (pa_id, _) in chunk_matrix if pa_id == f"PA{pa:02d}")
            for pa in range(1, 11)
        }
        chunks_per_dimension = {
            f"DIM{dim:02d}": sum(
                1 for (_, dim_id) in chunk_matrix if dim_id == f"DIM{dim:02d}"
            )
            for dim in range(1, 7)
        }

        logger.info(
            "chunk_matrix_constructed",
            extra={
                "total_chunks": len(chunk_matrix),
                "inserted_count": inserted_count,
                "chunks_per_policy_area": chunks_per_policy_area,
                "chunks_per_dimension": chunks_per_dimension,
            },
        )

        return chunk_matrix

    def _compute_integrity_hash(self) -> str:
        payload = []
        for (pa_id, dim_id) in self._matrix_keys_sorted:
            chunk = self._chunk_matrix[(pa_id, dim_id)]
            text_hash = hashlib.sha256(chunk.text.encode("utf-8")).hexdigest()
            payload.append(
                {
                    "policy_area_id": pa_id,
                    "dimension_id": dim_id,
                    "chunk_id": chunk.chunk_id,
                    "text_sha256": text_hash,
                }
            )

        json_bytes = json.dumps(payload, sort_keys=True, separators=(",", ":")).encode("utf-8")
        return hashlib.sha256(json_bytes).hexdigest()


__all__ = ["ChunkMatrix", "SmartPolicyChunk"]



################################################################################
# FILE: phase2_v_executor_chunk_synchronizer.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_v_executor_chunk_synchronizer.py
################################################################################

"""
Module: phase2_v_executor_chunk_synchronizer
PHASE_LABEL: Phase 2
Sequence: V
Description: Executor-chunk JOIN table synchronization

Version: 1.0.0
Last Modified: 2025-12-20
Author: F.A.R.F.A.N Policy Pipeline
License: Proprietary

This module is part of Phase 2: Analysis & Question Execution.
All files in Phase_two/ must contain PHASE_LABEL: Phase 2.
```

Phase 2 Source Code

```
"""
from __future__ import annotations

import json
import logging
from dataclasses import dataclass, field
from datetime import datetime
from pathlib import Path
from typing import Any, Literal

logger = logging.getLogger(__name__)

# Constants
EXPECTED_CONTRACT_COUNT = 300  # Q001-Q300
EXPECTED_CHUNK_COUNT = 60      # 10 PA Ã\227 6 DIM
DEFAULT_CONTRACT_DIR = "config/executor_contracts/specialized"


def _extract_chunk_coordinates(chunk: Any) -> tuple[str | None, str | None]:
    """Extract policy_area_id and dimension_id from a chunk.

    Supports both object attributes and dict keys for flexibility.

    Args:
        chunk: Chunk object or dict with coordinates

    Returns:
        Tuple of (policy_area_id, dimension_id) or (None, None) if not found
    """
    policy_area_id = getattr(chunk, "policy_area_id", None)
    if policy_area_id is None and isinstance(chunk, dict):
        policy_area_id = chunk.get("policy_area_id")

    dimension_id = getattr(chunk, "dimension_id", None)
    if dimension_id is None and isinstance(chunk, dict):
        dimension_id = chunk.get("dimension_id")

    return policy_area_id, dimension_id


class ExecutorChunkSynchronizationError(Exception):
    """Raised when executor-chunk synchronization fails.

    This exception indicates a violation of synchronization invariants:
    - Missing chunks for executor contracts
    - Duplicate chunks for the same (PA, DIM) coordinates
    - Routing key mismatches
    - 1:1 mapping violations
    """
    pass


@dataclass
class ExecutorChunkBinding:
    """Canonical JOIN table entry: 1 executor contract â\206\222 1 chunk.

    Constitutional Invariants:
    - Each executor_contract_id maps to exactly 1 chunk_id
    - Each chunk_id maps to exactly 1 executor_contract_id
    - Total bindings = 300 (all Q001-Q300 contracts)

    Attributes:
        executor_contract_id: Contract identifier (Q001-Q300)
        policy_area_id: Policy area identifier (PA01-PA10)
        dimension_id: Dimension identifier (DIM01-DIM06)
        chunk_id: Chunk identifier ("PA01-DIM01" format) or None if missing
        chunk_index: Position in chunk list or None if missing
        expected_patterns: Patterns from contract.question_context.patterns
        irrigated_patterns: Actual patterns delivered to chunk
        pattern_count: Number of expected patterns
        expected_signals: Required signals from contract.signal_requirements
        irrigated_signals: Actual signal instances delivered
        signal_count: Number of irrigated signals
        status: Binding status (matched, missing_chunk, duplicate_chunk, mismatch, missing_signals)
        contract_file: Path to contract JSON file
        contract_hash: SHA-256 from contract.identity.contract_hash
        chunk_source: Source of chunk data (typically "phase1_spc_ingestion")
        validation_errors: List of error messages
        validation_warnings: List of warning messages
    """

    # Identity
    executor_contract_id: str
    policy_area_id: str
    dimension_id: str

    # Routing
    chunk_id: str | None
    chunk_index: int | None

    # Pattern Irrigation
    expected_patterns: list[dict[str, Any]]
    irrigated_patterns: list[dict[str, Any]]
    pattern_count: int

    # Signal Irrigation
    expected_signals: list[str]
    irrigated_signals: list[dict[str, Any]]
    signal_count: int

    # Status
    status: Literal[
        "matched",           # â\234\205 1:1 binding successful
        "missing_chunk",     # â\235\214 No chunk found for (PA, DIM)
        "duplicate_chunk",   # â\235\214 Multiple chunks match (PA, DIM)
        "mismatch",          # â\235\214 Routing key inconsistency
        "missing_signals"    # â\235\214 Required signals not delivered
    ]

    # Provenance
    contract_file: str
    contract_hash: str
    chunk_source: str

    # Validation
    validation_errors: list[str] = field(default_factory=list)
    validation_warnings: list[str] = field(default_factory=list)

    def to_dict(self) -> dict[str, Any]:
        """Convert binding to dictionary for serialization."""
        return {
            "executor_contract_id": self.executor_contract_id,
            "policy_area_id": self.policy_area_id,
            "dimension_id": self.dimension_id,
            "chunk_id": self.chunk_id,
            "chunk_index": self.chunk_index,
            "patterns_expected": self.pattern_count,
            "patterns_delivered": len(self.irrigated_patterns),
            "pattern_ids": [p.get("id", "UNKNOWN") for p in self.irrigated_patterns],
            "signals_expected": len(self.expected_signals),
            "signals_delivered": self.signal_count,
            "signal_types": [s.get("signal_type", "UNKNOWN") for s in self.irrigated_signals],
            "status": self.status,
            "provenance": {
                "contract_file": self.contract_file,
                "contract_hash": self.contract_hash,
                "chunk_source": self.chunk_source,
                "chunk_index": self.chunk_index
            },
            "validation": {
                "errors": self.validation_errors,
                "warnings": self.validation_warnings
            }
        }


def build_join_table(
    contracts: list[dict[str, Any]],
    chunks: list[Any]
) -> list[ExecutorChunkBinding]:
    """Build canonical JOIN table with BLOCKING validation.

    Algorithm:
    1. For each contract in contracts:
        a. Extract (policy_area_id, dimension_id) from contract.identity
        b. Search chunks for matching (policy_area_id, dimension_id)
        c. If 0 matches â\206\222 status="missing_chunk", ABORT
        d. If 2+ matches â\206\222 status="duplicate_chunk", ABORT
        e. If 1 match â\206\222 status="matched", continue

    2. Validate 1:1 invariants:
        a. Each contract_id appears exactly once
        b. Each chunk_id appears exactly once
        c. Total bindings = 300

    3. Populate pattern and signal irrigation:
        a. Extract expected_patterns from contract.question_context.patterns
        b. Extract expected_signals from contract.signal_requirements
        c. Initialize irrigated_* fields (populated later by irrigation phase)

    4. Return binding table OR raise ExecutorChunkSynchronizationError

    Args:
        contracts: List of 300 executor contracts (Q001-Q300.v3.json)
        chunks: List of chunks from Phase 1 (should be 60 chunks)

    Returns:
        List of ExecutorChunkBinding objects (300 bindings)

    Raises:
        ExecutorChunkSynchronizationError: If any binding fails validation
    """
    bindings: list[ExecutorChunkBinding] = []

    logger.info(f"Building JOIN table: {len(contracts)} contracts Ã\227 {len(chunks)} chunks")

    for contract in contracts:
        # Extract identity from contract
        identity = contract.get("identity", {})
        contract_id = identity.get("question_id", "UNKNOWN")
        policy_area_id = identity.get("policy_area_id", "UNKNOWN")
        dimension_id = identity.get("dimension_id", "UNKNOWN")
        contract_hash = identity.get("contract_hash", "")

        # Find matching chunks
        matching_chunks = []
        for i, chunk in enumerate(chunks):
            chunk_pa, chunk_dim = _extract_chunk_coordinates(chunk)

            if chunk_pa == policy_area_id and chunk_dim == dimension_id:
                matching_chunks.append((i, chunk))

        # Validate 1:1 mapping
        if len(matching_chunks) == 0:
            # ABORT: No chunk found
            error_msg = (
                f"No chunk found for {contract_id} with "
                f"PA={policy_area_id}, DIM={dimension_id}"
            )
            logger.error(error_msg)
            raise ExecutorChunkSynchronizationError(error_msg)

        if len(matching_chunks) > 1:
            # ABORT: Duplicate chunks
            error_msg = (
                f"Duplicate chunks for {contract_id}: found {len(matching_chunks)} chunks "
                f"with PA={policy_area_id}, DIM={dimension_id}"
            )
            logger.error(error_msg)
            raise ExecutorChunkSynchronizationError(error_msg)

        # Extract single matching chunk
        chunk_index, chunk = matching_chunks[0]
        raw_chunk_id = (
            getattr(chunk, "chunk_id", None)
            or (chunk.get("chunk_id") if isinstance(chunk, dict) else None)
            or f"{policy_area_id}-{dimension_id}"
        )
        # Guarantee uniqueness per *binding*: multiple executor contracts may map to the same
        # underlying chunk, but each binding must have a unique identifier.
        chunk_id = f"{raw_chunk_id}-{contract_id}"

        # NOTE: chunk_id reuse is allowed: many contracts may map to the same PAÃ\227DIM chunk.

        # Extract patterns from contract (NOT from generic PA pack)
        question_context = contract.get("question_context", {})
        expected_patterns = question_context.get("patterns", [])

        # Extract signals from contract
        signal_requirements = contract.get("signal_requirements", {})
        expected_signals = signal_requirements.get("mandatory_signals", [])

        # Determine contract file path
        contract_file = f"{DEFAULT_CONTRACT_DIR}/{contract_id}.v3.json"

        # Create binding
        binding = ExecutorChunkBinding(
            executor_contract_id=contract_id,
            policy_area_id=policy_area_id,
            dimension_id=dimension_id,
            chunk_id=chunk_id,
            chunk_index=chunk_index,
            expected_patterns=expected_patterns,
            irrigated_patterns=[],  # Populated by irrigation phase
            pattern_count=len(expected_patterns),
            expected_signals=expected_signals,
            irrigated_signals=[],  # Populated by irrigation phase
            signal_count=0,
            status="matched",
            contract_file=contract_file,
            contract_hash=contract_hash,
            chunk_source="phase1_spc_ingestion",
            validation_errors=[],
            validation_warnings=[]
        )

        bindings.append(binding)

        logger.debug(
            f"Bound {contract_id} â\206\222 {chunk_id} "
            f"(patterns={len(expected_patterns)}, signals={len(expected_signals)})"
        )

    # Validate total bindings = EXPECTED_CONTRACT_COUNT
    if len(bindings) != EXPECTED_CONTRACT_COUNT:
        error_msg = f"Expected 300 bindings, got {len(bindings)}"
        logger.error(error_msg)
        raise ExecutorChunkSynchronizationError(error_msg)

    # Validate uniqueness
    validate_uniqueness(bindings)

    logger.info(f"â\234\223 JOIN table built successfully: {len(bindings)} bindings")

    return bindings


def validate_uniqueness(bindings: list[ExecutorChunkBinding]) -> None:
```

Phase 2 Source Code

```
    """Validate binding invariants.

    Checks:
    1. Each contract_id appears exactly once
    2. Each binding chunk_id appears exactly once
    3. Total bindings = 300

    Args:
        bindings: List of ExecutorChunkBinding objects

    Raises:
        ExecutorChunkSynchronizationError: If any invariant is violated
    """
    # Check each contract_id appears exactly once
    contract_ids = [b.executor_contract_id for b in bindings]
    if len(contract_ids) != len(set(contract_ids)):
        duplicates = [cid for cid in contract_ids if contract_ids.count(cid) > 1]
        unique_duplicates = list(set(duplicates))
        error_msg = f"Duplicate executor_contract_ids: {unique_duplicates}"
        logger.error(error_msg)
        raise ExecutorChunkSynchronizationError(error_msg)

    # Check each binding chunk_id appears exactly once
    chunk_ids = [b.chunk_id for b in bindings if b.chunk_id]
    if len(chunk_ids) != len(set(chunk_ids)):
        duplicates = [cid for cid in chunk_ids if chunk_ids.count(cid) > 1]
        unique_duplicates = list(set(duplicates))
        error_msg = f"Duplicate chunk_ids: {unique_duplicates}"
        logger.error(error_msg)
        raise ExecutorChunkSynchronizationError(error_msg)

    # Check total bindings = EXPECTED_CONTRACT_COUNT
    if len(bindings) != EXPECTED_CONTRACT_COUNT:
        error_msg = f"Expected {EXPECTED_CONTRACT_COUNT} bindings, got {len(bindings)}"
        logger.error(error_msg)
        raise ExecutorChunkSynchronizationError(error_msg)

    logger.debug("â\234\223 Binding invariants validated")


def generate_verification_manifest(
    bindings: list[ExecutorChunkBinding],
    include_full_bindings: bool = True
) -> dict[str, Any]:
    """Generate binding-specific verification manifest.

    Creates a comprehensive manifest with:
    - Binding details for all 300 contracts
    - Invariant validation results
    - Statistics on patterns, signals, and coverage
    - Error and warning aggregation

    Args:
        bindings: List of ExecutorChunkBinding objects
        include_full_bindings: If True, include full binding details (default: True)

    Returns:
        Dictionary with manifest data ready for JSON serialization
    """
    # Aggregate errors and warnings
    all_errors = [e for b in bindings for e in b.validation_errors]
    all_warnings = [w for b in bindings for w in b.validation_warnings]

    # Count bindings by status
    bindings_by_status = {}
    for b in bindings:
        bindings_by_status[b.status] = bindings_by_status.get(b.status, 0) + 1

    # Calculate statistics
    total_patterns_expected = sum(b.pattern_count for b in bindings)
    total_patterns_delivered = sum(len(b.irrigated_patterns) for b in bindings)
    total_signals_expected = sum(len(b.expected_signals) for b in bindings)
    total_signals_delivered = sum(b.signal_count for b in bindings)

    avg_patterns = total_patterns_expected / len(bindings) if bindings else 0
    avg_signals = total_signals_expected / len(bindings) if bindings else 0

    # Validate invariants
    contract_ids = [b.executor_contract_id for b in bindings]
    chunk_ids = [b.chunk_id for b in bindings if b.chunk_id]

    invariants_validated = {
        # Historical key expected by tests.
        "one_to_one_mapping": (len(contract_ids) == len(set(contract_ids))) and (len(chunk_ids) == len(set(chunk_ids))),
        "all_contracts_have_chunks": all(b.chunk_id is not None for b in bindings),
        "all_chunks_assigned": all(b.status == "matched" for b in bindings),
        "no_duplicate_irrigation": len(chunk_ids) == len(set(chunk_ids)),
        "total_bindings_equals_expected": len(bindings) == EXPECTED_CONTRACT_COUNT,
    }

    # Build manifest
    manifest: dict[str, Any] = {
        "version": "1.0.0",
        "success": len(all_errors) == 0,
        "timestamp": datetime.utcnow().isoformat() + "Z",
        "total_contracts": len(bindings),
        # Count unique PAÃ\227DIM chunks (not per-binding chunk_id).
        "total_chunks": len({(b.policy_area_id, b.dimension_id) for b in bindings if b.chunk_id}),
        "errors": all_errors,
        "warnings": all_warnings,
        "invariants_validated": invariants_validated,
        "statistics": {
            "avg_patterns_per_binding": round(avg_patterns, 2),
            "avg_signals_per_binding": round(avg_signals, 2),
            "total_patterns_expected": total_patterns_expected,
            "total_patterns_delivered": total_patterns_delivered,
            "total_signals_expected": total_signals_expected,
            "total_signals_delivered": total_signals_delivered,
            "bindings_by_status": bindings_by_status
        }
    }

    # Include full binding details if requested
    if include_full_bindings:
        manifest["bindings"] = [b.to_dict() for b in bindings]

    return manifest


def save_verification_manifest(
    manifest: dict[str, Any],
    output_path: Path | str
) -> None:
    """Save verification manifest to JSON file.

    Args:
        manifest: Manifest dictionary from generate_verification_manifest()
        output_path: Path to output JSON file
    """
    output_path = Path(output_path)
    output_path.parent.mkdir(parents=True, exist_ok=True)

    with open(output_path, "w", encoding="utf-8") as f:
        json.dump(manifest, f, indent=2, ensure_ascii=False)

    logger.info(f"â\234\223 Verification manifest saved to {output_path}")


def load_executor_contracts(contracts_dir: Path | str) -> list[dict[str, Any]]:
    """Load all executor contracts from directory.

    Args:
        contracts_dir: Path to directory containing Q{nnn}.v3.json files

    Returns:
        List of contract dictionaries (Q001-Q300)

    Raises:
        FileNotFoundError: If contracts directory doesn't exist
        ValueError: If contract count != 300
    """
    contracts_dir = Path(contracts_dir)

    if not contracts_dir.exists():
        raise FileNotFoundError(f"Contracts directory not found: {contracts_dir}")

    contracts: list[dict[str, Any]] = []

    for i in range(1, EXPECTED_CONTRACT_COUNT + 1):
        contract_id = f"Q{i:03d}"
        contract_path = contracts_dir / f"{contract_id}.v3.json"

        if not contract_path.exists():
            logger.warning(f"Contract not found: {contract_path}")
            continue

        with open(contract_path, "r", encoding="utf-8") as f:
            contract = json.load(f)
            contracts.append(contract)

    if len(contracts) != EXPECTED_CONTRACT_COUNT:
        raise ValueError(
            f"Expected {EXPECTED_CONTRACT_COUNT} contracts, found {len(contracts)} in {contracts_dir}"
        )

    logger.info(f"â\234\223 Loaded {len(contracts)} executor contracts from {contracts_dir}")

    return contracts


################################################################################
# FILE: phase2_w_factory.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_w_factory.py
################################################################################

"""
Module: phase2_w_factory
PHASE_LABEL: Phase 2
Sequence: W

"""
```

Phase 2 Source Code

~~~
"""
Factory module â\200\224 canonical Dependency Injection (DI) and access control for F.A.R.F.A.N.

This module is the SINGLE AUTHORITATIVE BOUNDARY for:
- Canonical monolith access (CanonicalQuestionnaire) - loaded ONCE with integrity verification
- Signal registry construction (QuestionnaireSignalRegistry v2.0) from canonical source ONLY
- Method injection via MethodExecutor with signal registry DI
- Orchestrator construction with full DI (questionnaire, method_executor, executor_config)
- EnrichedSignalPack creation and injection per executor (30 executors)
- Hard contracts and validation constants for Phase 1
- SeedRegistry singleton initialization for determinism

METHOD DISPENSARY PATTERN - Core Architecture:
==============================================

The pipeline uses a "method dispensary" pattern where monolithic analyzer classes
serve as "dispensaries" that provide methods to executors. This architecture enables:

1. LOOSE COUPLING: Executors orchestrate methods without direct imports
2. PARTIAL REUSE: Same method used by multiple executors with different contexts
3. CENTRALIZED MANAGEMENT: All method routing through MethodExecutor with validation
4. SIGNAL AWARENESS: Methods receive signal packs for pattern matching

Dispensary Registry (~20 monolith classes, 240+ methods):
-------------------------------------------------------
- IndustrialPolicyProcessor (17 methods): Pattern matching, evidence extraction
- PDETMunicipalPlanAnalyzer (52+ methods): LARGEST - financial, causal, entity analysis
- CausalExtractor (28 methods): Goal extraction, causal hierarchy, semantic distance
- FinancialAuditor (13 methods): Budget tracing, allocation gaps, sufficiency
- BayesianMechanismInference (14 methods): Necessity/sufficiency tests, coherence
- BayesianCounterfactualAuditor (9 methods): SCM construction, refutation
- TextMiningEngine (8 methods): Critical link diagnosis, intervention generation
- SemanticAnalyzer (12 methods): Semantic cube, domain classification
- PerformanceAnalyzer (5 methods): Performance metrics, loss functions
- PolicyContradictionDetector (8 methods): Contradiction detection, coherence
- [... 10+ more classes]

Executor Usage Pattern:
----------------------
Each of 30 executors uses a UNIQUE COMBINATION of methods:
- D1-Q1 (QuantitativeBaselineExtractor): 17 methods from 9 classes
- D3-Q2 (TargetProportionalityAnalyzer): 24 methods from 7 classes
- D3-Q5 (OutputOutcomeLinkageAnalyzer): 28 methods from 6 classes
- D6-Q3 (ValidationTestingAnalyzer): 8 methods from 4 classes

Methods are orchestrated via:
```python
result = self.method_executor.execute(
    class_name="PDETMunicipalPlanAnalyzer",
    method_name="_score_indicators",
    document=doc,
    signal_pack=pack,
    **context
)
~~~

NOT ALL METHODS ARE USED:
- Monoliths contain more methods than executors need
- Only methods in executors_methods.json are actively used
- Phase 1 (ingestion) uses additional methods not in executor contracts
- 14 methods in validation failures (deprecated/private)

Design Principles (Factory Pattern + DI):
===

1. FACTORY PATTERN: AnalysisPipelineFactory is the ONLY place that instantiates:
   - Orchestrator, MethodExecutor, QuestionnaireSignalRegistry, BaseExecutor instances
   - NO other module should directly instantiate these classes

2. DEPENDENCY INJECTION: All components receive dependencies via __init__:
   - Orchestrator receives: questionnaire, method_executor, executor_config, validation_constants
   - MethodExecutor receives: method_registry, arg_router, signal_registry
   - BaseExecutor (30 classes) receive: enriched_signal_pack, method_executor, config

3. CANONICAL MONOLITH CONTROL:
   - load_questionnaire() called ONCE by factory only (singleton + integrity hash)
   - Orchestrator uses self.questionnaire object, NEVER file paths
   - Search codebase: NO other load_questionnaire() calls should exist

4. SIGNAL REGISTRY CONTROL:
   - create_signal_registry(questionnaire) - from canonical source ONLY
   - signal_loader.py MUST BE DELETED (legacy JSON loaders eliminated)
   - Registry injected into MethodExecutor, NOT accessed globally

5. ENRICHED SIGNAL PACK INJECTION:
   - Factory builds EnrichedSignalPack per executor (semantic expansion + context filtering)
   - Each BaseExecutor receives its specific pack, NOT full registry

6. DETERMINISM:
   - SeedRegistry singleton initialized by factory for reproducibility
   - ExecutorConfig encapsulates operational params (max_tokens, retries)

7. PHASE 1 HARD CONTRACTS:
   - Validation constants (P01_EXPECTED_CHUNK_COUNT=60, etc.) loaded by factory
   - Injected into Orchestrator for Phase 1 chunk validation
   - Execution FAILS if contracts violated
```
```

Phase 2 Source Code

```
SIN_CARRETA Compliance:
- All construction paths emit structured telemetry with timestamps and hashes
- Determinism enforced via explicit validation of canonical questionnaire integrity
- Contract assertions guard all factory outputs (no silent degradation)
- Auditability via immutable ProcessorBundle with provenance metadata
"""

from __future__ import annotations

import hashlib
import json
import logging
from datetime import datetime, timezone
from pathlib import Path
import time
from collections.abc import Mapping
from dataclasses import dataclass, field
from typing import Any, TYPE_CHECKING

# Phase 2 orchestration components
from canonic_phases.Phase_two.arg_router import ExtendedArgRouter
from orchestration.class_registry import build_class_registry, get_class_paths
from canonic_phases.Phase_two.executors.executor_config import ExecutorConfig
from canonic_phases.Phase_two.executors.base_executor_with_contract import BaseExecutorWithContract

# Core orchestration
if TYPE_CHECKING:
    from orchestration.orchestrator import MethodExecutor, Orchestrator
from orchestration.method_registry import (
    MethodRegistry,
    setup_default_instantiation_rules,
)

# SISAS - Signal Intelligence Layer (Nivel 2)
from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_intelligence_layer import (
    EnrichedSignalPack,
    create_enriched_signal_pack,
)
from cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_registry import (
    QuestionnaireSignalRegistry,
    create_signal_registry,
)

# Phase 1 validation constants module
# NOTE: validation_constants module does not exist in current architecture
# Using empty fallback - implement in future JOBFRONT if needed
PHASE1_VALIDATION_CONSTANTS: dict[str, Any] = {}
VALIDATION_CONSTANTS_AVAILABLE = False

def load_validation_constants() -> dict[str, Any]:
    """Stub for validation constants loading (module not yet implemented)."""
    return PHASE1_VALIDATION_CONSTANTS

# Optional: CoreModuleFactory for I/O helpers
# NOTE: CoreModuleFactory does not exist in current architecture
CoreModuleFactory = None
CORE_MODULE_FACTORY_AVAILABLE = False

# SeedRegistry for determinism
from orchestration.seed_registry import SeedRegistry
SEED_REGISTRY_AVAILABLE = True

logger = logging.getLogger(__name__)


# ============================================================================
# RUTA CANÃ\223NICA DEL CUESTIONARIO - NIVEL 1
# SegÃºn AGENTS.md - NO MODIFICAR sin actualizar documentaciÃ³n
# ============================================================================
_REPO_ROOT = Path(__file__).resolve().parents[2]
CANONICAL_QUESTIONNAIRE_PATH = _REPO_ROOT / "canonic_questionnaire_central" / "questionnaire_monolith.json"


@dataclass(frozen=True)
class CanonicalQuestionnaire:
    """
    Objeto inmutable del cuestionario monolito.

    NIVEL 1: Acceso Total
    CONSUMIDOR Ã\232NICO: AnalysisPipelineFactory (este archivo)
    PROHIBIDO: Instanciar directamente, usar load_questionnaire()
    """
    data: dict[str, Any]
    sha256: str
    version: str
    load_timestamp: str
    source_path: str

    @property
    def dimensions(self) -> dict[str, Any]:
        """6 dimensiones: DIM01-DIM06"""
        return dict(self.data.get("canonical_notation", {}).get("dimensions", {}))

    @property
    def policy_areas(self) -> dict[str, Any]:
        """10 Ã¡reas: PA01-PA10"""
        return dict(self.data.get("canonical_notation", {}).get("policy_areas", {}))

    @property
    def micro_questions(self) -> list[dict[str, Any]]:
        """300 micro preguntas"""
        return list(self.data.get("blocks", {}).get("micro_questions", []))

    @property
    def meso_questions(self) -> list[dict[str, Any]]:
        """4 meso preguntas"""
        return list(self.data.get("blocks", {}).get("meso_questions", []))

    @property
    def macro_question(self) -> dict[str, Any]:
        """1 macro pregunta"""
        return dict(self.data.get("blocks", {}).get("macro_question", {}))


class QuestionnaireLoadError(Exception):
    """Error al cargar el cuestionario."""
    pass


class QuestionnaireIntegrityError(QuestionnaireLoadError):
    """Hash del cuestionario no coincide."""
    pass


def load_questionnaire(
    path: Path | None = None,
    expected_hash: str | None = None,
) -> CanonicalQuestionnaire:
    """
    Carga el cuestionario canÃ³nico con verificaciÃ³n de integridad.

    NIVEL 1: Ã\232NICA funciÃ³n autorizada para I/O del monolito.
    CONSUMIDOR: Solo AnalysisPipelineFactory._load_canonical_questionnaire

    Args:
        path: Ruta al archivo (default: CANONICAL_QUESTIONNAIRE_PATH)
        expected_hash: Hash SHA256 esperado para verificaciÃ³n

    Returns:
        CanonicalQuestionnaire: Objeto inmutable verificado

    Raises:
        QuestionnaireLoadError: Archivo no existe o JSON invÃ¡lido
        QuestionnaireIntegrityError: Hash no coincide
    """
    questionnaire_path = path or CANONICAL_QUESTIONNAIRE_PATH

    if not questionnaire_path.exists():
        raise QuestionnaireLoadError(
            f"Questionnaire not found: {questionnaire_path}"
        )

    content_bytes = questionnaire_path.read_bytes()
    computed_hash = hashlib.sha256(content_bytes).hexdigest()

    if expected_hash and computed_hash.lower() != expected_hash.lower():
        raise QuestionnaireIntegrityError(
            f"Hash mismatch: expected {expected_hash[:16]}..., "
            f"got {computed_hash[:16]}..."
        )

    try:
        content = json.loads(content_bytes.decode("utf-8"))
    except json.JSONDecodeError as e:
        raise QuestionnaireLoadError(f"Invalid JSON: {e}")

    if "canonical_notation" not in content:
        raise QuestionnaireLoadError("Missing 'canonical_notation'")
    if "blocks" not in content:
        raise QuestionnaireLoadError("Missing 'blocks'")

    version = content.get("version", "unknown")

    return CanonicalQuestionnaire(
        data=content,
        sha256=computed_hash,
        version=version,
        load_timestamp=datetime.now(timezone.utc).isoformat(),
        source_path=str(questionnaire_path.resolve()),
    )


# =============================================================================
# Exceptions
# =============================================================================


class FactoryError(Exception):
    """Base exception for factory construction failures."""
    pass
```

Phase 2 Source Code

```
class QuestionnaireValidationError(FactoryError):
    """Raised when questionnaire validation fails."""
    pass


class IntegrityError(FactoryError):
    """Raised when questionnaire integrity check (SHA-256) fails."""
    pass


class RegistryConstructionError(FactoryError):
    """Raised when signal registry construction fails."""
    pass


class ExecutorConstructionError(FactoryError):
    """Raised when method executor construction fails."""
    pass


class SingletonViolationError(FactoryError):
    """Raised when singleton pattern is violated."""
    pass


# =============================================================================
# Processor Bundle (typed DI container with provenance)
# =============================================================================


@dataclass(frozen=True)
class ProcessorBundle:
    """Aggregated orchestrator dependencies built by the Factory.

    This is the COMPLETE DI container returned by AnalysisPipelineFactory.

    Attributes:
        orchestrator: Fully configured Orchestrator (main entry point).
        method_executor: MethodExecutor with signal registry injected.
        questionnaire: Immutable, validated CanonicalQuestionnaire (monolith).
        signal_registry: QuestionnaireSignalRegistry v2.0 from canonical source.
        executor_config: ExecutorConfig for operational parameters.
        enriched_signal_packs: Dict of EnrichedSignalPack per policy area.
        validation_constants: Phase 1 hard contracts (chunk counts, etc.).
        core_module_factory: Optional CoreModuleFactory for I/O helpers.
        seed_registry_initialized: Whether SeedRegistry singleton was set up.
        provenance: Construction metadata for audit trails.
    """

    orchestrator: Orchestrator
    method_executor: MethodExecutor
    questionnaire: CanonicalQuestionnaire
    signal_registry: QuestionnaireSignalRegistry
    executor_config: ExecutorConfig
    enriched_signal_packs: dict[str, EnrichedSignalPack]
    validation_constants: dict[str, Any]
    core_module_factory: Any | None = None
    seed_registry_initialized: bool = False
    provenance: dict[str, Any] = field(default_factory=dict)

    def __post_init__(self) -> None:
        """SIN_CARRETA Â§ Contract Enforcement: validate bundle integrity."""
        errors = []

        # Critical components validation
        if self.orchestrator is None:
            errors.append("orchestrator must not be None")
        if self.method_executor is None:
            errors.append("method_executor must not be None")
        if self.questionnaire is None:
            errors.append("questionnaire must not be None")
        if self.signal_registry is None:
            errors.append("signal_registry must not be None")
        if self.executor_config is None:
            errors.append("executor_config must not be None")
        if self.enriched_signal_packs is None:
            errors.append("enriched_signal_packs must not be None")
        elif not isinstance(self.enriched_signal_packs, dict):
            errors.append("enriched_signal_packs must be dict[str, EnrichedSignalPack]")

        if self.validation_constants is None:
            errors.append("validation_constants must not be None")

        # Provenance validation
        if not self.provenance.get("construction_timestamp_utc"):
            errors.append("provenance must include construction_timestamp_utc")
        if not self.provenance.get("canonical_sha256"):
            errors.append("provenance must include canonical_sha256")
        if self.provenance.get("signal_registry_version") != "2.0":
            errors.append("provenance must indicate signal_registry_version=2.0")

        # Factory pattern enforcement check
        if not self.provenance.get("factory_instantiation_confirmed"):
            errors.append("provenance must confirm factory instantiation (not direct construction)")
        if errors:
            raise FactoryError(f"ProcessorBundle validation failed: {'; '.join(errors)}")

        logger.info(
            "processor_bundle_validated "
            "canonical_sha256=%s construction_ts=%s policy_areas=%d validation_constants=%d",
            self.provenance.get("canonical_sha256", "")[:16],
            self.provenance.get("construction_timestamp_utc"),
            len(self.enriched_signal_packs),
            len(self.validation_constants),
        )


# =============================================================================
# Analysis Pipeline Factory (Main Factory Class)
# =============================================================================


class AnalysisPipelineFactory:
    """Factory for constructing the complete analysis pipeline.

    This is the ONLY class that should instantiate:
    - Orchestrator
    - MethodExecutor
    - QuestionnaireSignalRegistry
    - BaseExecutor instances (30 executor classes)

    CRITICAL: No other module should directly instantiate these classes.
    All dependencies are injected via constructor parameters.

    Usage:
        factory = AnalysisPipelineFactory(
            questionnaire_path="path/to/questionnaire.json",
            expected_hash="abc123...",
            seed=42
        )
        bundle = factory.create_orchestrator()
        orchestrator = bundle.orchestrator
    """

    # Singleton tracking for load_questionnaire() call
    _questionnaire_loaded = False
    _questionnaire_instance: CanonicalQuestionnaire | None = None

    def __init__(
        self,
        *,
        questionnaire_path: str | None = None,
        expected_questionnaire_hash: str | None = None,
        executor_config: ExecutorConfig | None = None,
        validation_constants: dict[str, Any] | None = None,
        enable_intelligence_layer: bool = True,
        seed_for_determinism: int | None = None,
        strict_validation: bool = True,
    ):
        """Initialize the Analysis Pipeline Factory.

        Args:
            questionnaire_path: Path to canonical questionnaire JSON.
            expected_questionnaire_hash: Expected SHA-256 hash for integrity check.
            executor_config: Custom executor configuration (if None, uses default).
            validation_constants: Phase 1 validation constants (if None, loads from config).
            enable_intelligence_layer: Whether to build enriched signal packs.
            seed_for_determinism: Seed for SeedRegistry singleton.
            strict_validation: If True, fail on any validation error.
        """
        self._questionnaire_path = questionnaire_path
        self._expected_hash = expected_questionnaire_hash
        self._executor_config = executor_config
        self._validation_constants = validation_constants
        self._enable_intelligence = enable_intelligence_layer
        self._seed = seed_for_determinism
        self._strict = strict_validation

        # Internal state (set during construction)
        self._canonical_questionnaire: CanonicalQuestionnaire | None = None
        self._signal_registry: QuestionnaireSignalRegistry | None = None
        self._method_executor: MethodExecutor | None = None
        self._enriched_packs: dict[str, EnrichedSignalPack] = {}

        logger.info(
            "factory_initialized questionnaire_path=%s intelligence_layer=%s seed=%s",
            questionnaire_path or "default",
            enable_intelligence_layer,
            seed_for_determinism is not None,
        )

    def create_orchestrator(self) -> ProcessorBundle:
        """Create fully configured Orchestrator with all dependencies injected.

        This is the PRIMARY ENTRY POINT for the factory.
        Returns a complete ProcessorBundle with Orchestrator ready to use.

        Returns:
            ProcessorBundle: Immutable bundle with all dependencies wired.
        Raises:
            QuestionnaireValidationError: If questionnaire validation fails.
            IntegrityError: If questionnaire hash doesn't match expected.
            RegistryConstructionError: If signal registry construction fails.
            ExecutorConstructionError: If method executor construction fails.
        """
        construction_start = time.time()
        timestamp_utc = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())

        logger.info("factory_create_orchestrator_start timestamp=%s", timestamp_utc)

        try:
            # Step 1: Load canonical questionnaire (ONCE, with integrity check)
            self._load_canonical_questionnaire()

            # Step 2: Build signal registry from canonical source
            self._build_signal_registry()

            # Step 3: Build enriched signal packs (intelligence layer)
            self._build_enriched_signal_packs()

            # Step 4: Initialize seed registry for determinism
            seed_initialized = self._initialize_seed_registry()

            # Step 5: Build method executor with signal registry DI
            self._build_method_executor()

            # Step 6: Load Phase 1 validation constants
            validation_constants = self._load_validation_constants()

            # Step 7: Get or create executor config
            executor_config = self._get_executor_config()

            # Step 8: Build orchestrator with full DI
            orchestrator = self._build_orchestrator(
                executor_config=executor_config,
                validation_constants=validation_constants,
            )

            # Step 9: Assemble provenance metadata
            construction_duration = time.time() - construction_start
            canonical_hash = self._compute_questionnaire_hash()

            provenance = {
                "construction_timestamp_utc": timestamp_utc,
                "canonical_sha256": canonical_hash,
                "signal_registry_version": "2.0",
                "intelligence_layer_enabled": self._enable_intelligence,
                "enriched_packs_count": len(self._enriched_packs),
                "validation_constants_count": len(validation_constants),
                "construction_duration_seconds": round(construction_duration, 3),
                "seed_registry_initialized": seed_initialized,
                "core_module_factory_available": CORE_MODULE_FACTORY_AVAILABLE,
                "strict_validation": self._strict,
                "factory_instantiation_confirmed": True,  # Critical for bundle validation
                "factory_class": "AnalysisPipelineFactory",
            }

            # Step 10: Build complete bundle
            bundle = ProcessorBundle(
                orchestrator=orchestrator,
                method_executor=self._method_executor,
                questionnaire=self._canonical_questionnaire,
                signal_registry=self._signal_registry,
                executor_config=executor_config,
                enriched_signal_packs=self._enriched_packs,
                validation_constants=validation_constants,
                core_module_factory=self._build_core_module_factory(),
                seed_registry_initialized=seed_initialized,
                provenance=provenance,
            )

            logger.info(
                "factory_create_orchestrator_complete duration=%.3fs hash=%s",
                construction_duration,
                canonical_hash[:16],
            )

            return bundle

        except Exception as e:
            logger.error("factory_create_orchestrator_failed error=%s", str(e), exc_info=True)
            raise FactoryError(f"Failed to create orchestrator: {e}") from e

    # ==========================================================================
    # Internal Construction Methods
    # ==========================================================================

    def _load_canonical_questionnaire(self) -> None:
        """Load canonical questionnaire with singleton enforcement and integrity check.

        CRITICAL REQUIREMENTS:
        1. This is the ONLY place in the codebase that calls load_questionnaire()
        2. Must enforce singleton pattern (only load once)
        3. Must verify SHA-256 hash for integrity
        4. Must raise IntegrityError if hash doesn't match
        Raises:
            SingletonViolationError: If load_questionnaire() already called.
            IntegrityError: If questionnaire hash doesn’t match expected.
            QuestionnaireValidationError: If questionnaire structure invalid.
        """
        # Enforce singleton pattern
        if AnalysisPipelineFactory._questionnaire_loaded:
            if AnalysisPipelineFactory._questionnaire_instance is not None:
                logger.info("questionnaire_singleton_reused using_cached_instance")
                self._canonical_questionnaire = AnalysisPipelineFactory._questionnaire_instance
                return
            else:
                raise SingletonViolationError(
                    "load_questionnaire() was called but instance is None. "
                    "This indicates a singleton pattern violation."
                )

        logger.info("questionnaire_loading_start path=%s", self._questionnaire_path or "default")

        try:
            # Load questionnaire (this should be the ONLY call in the entire codebase)
            questionnaire = load_questionnaire(self._questionnaire_path)

            # Mark singleton as loaded
            AnalysisPipelineFactory._questionnaire_loaded = True
            AnalysisPipelineFactory._questionnaire_instance = questionnaire

            # Compute integrity hash
            actual_hash = self._compute_questionnaire_hash_from_instance(questionnaire)

            # Verify integrity if expected hash provided
            if self._expected_hash is not None:
                if actual_hash != self._expected_hash:
                    raise IntegrityError(
                        f"Questionnaire integrity check FAILED. "
                        f"Expected: {self._expected_hash[:16]}... "
                        f"Actual: {actual_hash[:16]}... "
                        f"The canonical questionnaire may have been tampered with."
                    )
                logger.info("questionnaire_integrity_verified hash=%s", actual_hash[:16])
            else:
                logger.warning(
                    "questionnaire_integrity_not_verified no_expected_hash_provided "
                    "actual_hash=%s",
                    actual_hash[:16]
                )

            # Validate structure
            if not hasattr(questionnaire, ’questions’):
                if self._strict:
                    raise QuestionnaireValidationError("Questionnaire missing ’questions’ attribute")
                logger.warning("questionnaire_validation_warning missing_questions_attribute")

            questions = getattr(questionnaire, ’questions’, [])
            if not questions:
                if self._strict:
                    raise QuestionnaireValidationError("Questionnaire has no questions")
                logger.warning("questionnaire_validation_warning no_questions")

            self._canonical_questionnaire = questionnaire

            logger.info(
                "questionnaire_loaded_successfully questions=%d hash=%s singleton=established",
                len(questions),
                actual_hash[:16],
            )

        except Exception as e:
            if isinstance(e, (IntegrityError, SingletonViolationError, QuestionnaireValidationError)):
                raise
            raise QuestionnaireValidationError(f"Failed to load questionnaire: {e}") from e

    def _build_signal_registry(self) -> None:
        """Build signal registry from canonical questionnaire.

        CRITICAL REQUIREMENTS:
        1. Use create_signal_registry(questionnaire) ONLY
        2. Pass self._canonical_questionnaire as ONLY argument
        3. NO other signal loading methods allowed (signal_loader.py DELETED)

        Raises:
            RegistryConstructionError: If registry construction fails.
        """
        if self._canonical_questionnaire is None:
            raise RegistryConstructionError(
                "Cannot build signal registry: canonical questionnaire not loaded"
            )

        logger.info("signal_registry_building_start")

        try:
            # Build registry from canonical source ONLY
            registry = create_signal_registry(self._canonical_questionnaire)

            # Validate registry
            if not hasattr(registry, 'get_all_policy_areas'):
                if self._strict:
                    raise RegistryConstructionError("Registry missing required methods")
                logger.warning("registry_validation_warning missing_methods")

            policy_areas = registry.get_all_policy_areas() if hasattr(registry, 'get_all_policy_areas') else []

            self._signal_registry = registry

            logger.info(
                "signal_registry_built_successfully version=2.0 policy_areas=%d",
                len(policy_areas),
            )

        except Exception as e:
            if isinstance(e, RegistryConstructionError):
                raise
            raise RegistryConstructionError(f"Failed to build signal registry: {e}") from e

    def _build_enriched_signal_packs(self) -> None:
        """Build enriched signal packs for all policy areas.

        Each BaseExecutor receives its own EnrichedSignalPack (NOT full registry).
        Pack includes semantic expansion and context filtering.

        Raises:
            RegistryConstructionError: If pack construction fails in strict mode.
        """
        if not self._enable_intelligence:
            logger.info("enriched_packs_disabled intelligence_layer=off")
            self._enriched_packs = {}
            return

        if self._signal_registry is None:
            raise RegistryConstructionError(
                "Cannot build enriched packs: signal registry not built"
            )

        logger.info("enriched_packs_building_start")

        enriched_packs: dict[str, EnrichedSignalPack] = {}

        try:
            policy_areas = self._signal_registry.get_all_policy_areas() if hasattr(self._signal_registry, 'get_all_policy_areas
') else []

            if not policy_areas:
                logger.warning("enriched_packs_warning no_policy_areas_found")
                self._enriched_packs = enriched_packs
                return

            for policy_area_id in policy_areas:
                try:
                    # Get base pack from registry
                    base_pack = self._signal_registry.get(policy_area_id) if hasattr(self._signal_registry, 'get') else None

                    if base_pack is None:
                        logger.warning("base_pack_missing policy_area=%s", policy_area_id)
                        continue

                    base_metadata = getattr(base_pack, "metadata", {}) or {}
                    pattern_specs = base_metadata.get("pattern_specs", [])
                    if not isinstance(pattern_specs, list):
                        pattern_specs = []

                    # Create enriched pack (semantic expansion + context filtering)
                    # NOTE: EnrichedSignalPack expects dict-based pattern specs, not raw strings.
                    enriched_pack = create_enriched_signal_pack(
                        base_signal_pack={"patterns": pattern_specs},
                        enable_semantic_expansion=True,
                    )

                    enriched_packs[policy_area_id] = enriched_pack

                    logger.debug(
                        "enriched_pack_created policy_area=%s",
                        policy_area_id,
                    )

                except Exception as e:
                    msg = f"Failed to create enriched pack for {policy_area_id}: {e}"
                    if self._strict:
                        raise RegistryConstructionError(msg) from e
                    logger.error("enriched_pack_creation_failed policy_area=%s", policy_area_id, exc_info=True)

            self._enriched_packs = enriched_packs

            logger.info(
                "enriched_packs_built_successfully count=%d",
                len(enriched_packs),
            )

        except Exception as e:
            if isinstance(e, RegistryConstructionError):
                raise
            raise RegistryConstructionError(f"Failed to build enriched packs: {e}") from e
```

Phase 2 Source Code

~~~
    def _initialize_seed_registry(self) -> bool:
        """Initialize SeedRegistry singleton for deterministic operations.

        Returns:
            bool: True if seed registry was initialized, False otherwise.
        """
        if not SEED_REGISTRY_AVAILABLE:
            logger.warning("seed_registry_unavailable module_not_found determinism_not_guaranteed")
            return False

        if self._seed is None:
            logger.info("seed_registry_not_initialized no_seed_provided")
            return False

        try:
            SeedRegistry.initialize(master_seed=self._seed)
            logger.info("seed_registry_initialized master_seed=%d determinism=enabled", self._seed)
            return True
        except Exception:
            logger.error("seed_registry_initialization_failed", exc_info=True)
            return False

    def _build_method_executor(self) -> None:
        """Build MethodExecutor with full dependency wiring.

        CRITICAL INTEGRATION POINT - Method Dispensary Pattern:
        =========================================================

        This is where the "monolith dispensaries" get wired into the pipeline.
        The 30 executors orchestrate methods from these dispensaries WITHOUT
        direct imports or tight coupling to the monolith implementations.

        Architecture Flow:
        -----------------
        1. build_class_registry() loads the "method dispensaries" (monoliths):
           - IndustrialPolicyProcessor: 17 methods used across D1-Q1, D1-Q5, D2-Q2, D3-Q1
           - BayesianEvidenceScorer: 8 methods for confidence calculation
           - PDETMunicipalPlanAnalyzer: 52+ methods (LARGEST dispensary)
             Used in: D1-Q2, D1-Q3, D1-Q4, D2-Q1, D3-Q2, D3-Q3, D3-Q4, D3-Q5,
                      D4-Q1, D4-Q2, D4-Q3, D5-Q1, D5-Q2, D5-Q4, D5-Q5
           - CausalExtractor: 28 methods for causal inference
           - FinancialAuditor: 13 methods for financial analysis
           - BayesianMechanismInference: 14 methods for mechanism testing
           - [... 15+ more classes from farfan_core]

           Total: ~240 method pairs validated (see executor_factory_validation.json)

        2. These classes are NOT instantiated here - they’re registered as TYPES.
           Instantiation happens lazily via MethodRegistry when methods are called.

        3. ExtendedArgRouter receives the class registry and provides:
           - 30+ special routes for high-traffic methods (see arg_router.py)
           - Generic routing via signature inspection for all other methods
           - Strict argument validation (no silent parameter drops)
           - **kwargs awareness for forward compatibility

        4. MethodExecutor combines three critical components:
           - MethodRegistry: Instantiation rules + shared instances (e.g., MunicipalOntology)
           - ArgRouter: Method routing + argument validation
           - SignalRegistry: Injected for signal-aware methods

        5. Each of the 30 Executors orchestrates methods via:
           ```python
           result = self.method_executor.execute(
               class_name="PDETMunicipalPlanAnalyzer",
               method_name="_score_indicators",
               **payload  # document, question_id, signal_pack, etc.
           )
           ```

        Method Reuse Pattern:
        --------------------
        - Methods are PARTIALLY reused across executors (not fully shared)
        - Example: "_score_indicators" used in D3-Q1, D3-Q2, D4-Q1
        - Example: "_test_sufficiency" used in D2-Q2, D3-Q2, D3-Q4
        - Each executor uses a DIFFERENT COMBINATION of methods
        - Total unique combinations: 30 executors Ã\227 avg 12 methods = ~360 method calls

        Not All Methods Are Used:
        ------------------------
        The monoliths contain MORE methods than executors need.
        Only methods listed in executors_methods.json are actively used.
        Phase 1 (ingestion) uses additional methods not in executor contracts.

        Validation:
        ----------
        - executor_factory_validation.json: 243 pairs validated, 14 failures
        - Failures are methods NOT in catalog (likely private/deprecated)
        - All executor contracts reference validated methods only

        Signal Registry Integration:
        ---------------------------
        Signal registry is injected so methods can access:
        - Policy-area-specific patterns
        - Expected elements for validation
~~~

```
        - Semantic enrichment via intelligence layer

        Raises:
            ExecutorConstructionError: If executor construction fails.

        See Also:
            - executors_methods.json: Complete executorâ\206\222methods mapping
            - executor_factory_validation.json: Method catalog validation
            - arg_router.py: Special routes and routing logic
            - class_registry.py: Monolith class paths (_CLASS_PATHS)
        """
        if self._signal_registry is None:
            raise ExecutorConstructionError(
                "Cannot build method executor: signal registry not built"
            )

        logger.info("method_executor_building_start dispensaries=loading")

        try:
            # Step 1: Build method registry with special instantiation rules
            # MethodRegistry handles shared instances (e.g., MunicipalOntology singleton)
            # and custom instantiation logic for complex analyzers
            method_registry = MethodRegistry()
            setup_default_instantiation_rules(method_registry)

            logger.info("method_registry_built instantiation_rules=configured")

            # Step 2: Build class registry - THE METHOD DISPENSARIES
            # This loads ~20 monolith classes with 240+ methods total
            # Each class is a "dispensary" that provides methods to executors
            class_registry = build_class_registry()

            logger.info(
                "class_registry_built dispensaries=%d total_methods=240+",
                len(class_registry)
            )

            # Step 3: Build extended arg router with special routes
            # Handles 30+ high-traffic method routes + generic routing
            arg_router = ExtendedArgRouter(class_registry)

            special_routes = arg_router.get_special_route_coverage() if hasattr(arg_router, 'get_special_route_coverage') else
0

            logger.info(
                "arg_router_built special_routes=%d generic_routing=enabled",
                special_routes
            )

            # Step 4: Build method executor WITH signal registry injected
            # This is the CORE integration point - executors call methods through this
            # Local import to avoid circular dependency
            from orchestration.orchestrator import MethodExecutor
            method_executor = MethodExecutor(
                method_registry=method_registry,
                arg_router=arg_router,
                signal_registry=self._signal_registry,  # DI: inject signal registry
            )

            # Step 5: PRE-EXECUTION CONTRACT VERIFICATION
            # Verify all 30 base executor contracts (D1-Q1 through D6-Q5) before execution
            # This ensures contract integrity and method class availability at startup
            logger.info("contract_verification_start verifying_30_base_contracts")



            verification_result = BaseExecutorWithContract.verify_all_base_contracts(
                class_registry=class_registry
            )

            if not verification_result["passed"]:
                error_summary = f"{len(verification_result['errors'])} contract validation errors"
                logger.error(
                    "contract_verification_failed errors=%d warnings=%d",
                    len(verification_result["errors"]),
                    len(verification_result.get("warnings", [])),
                )

                for error in verification_result["errors"][:10]:
                    logger.error("contract_error: %s", error)

                if self._strict:
                    raise ExecutorConstructionError(
                        f"Pre-execution contract verification failed: {error_summary}. "
                        f"See logs for details. Total errors: {len(verification_result['errors'])}"
                    )
                else:
                    logger.warning(
                        "contract_verification_failed_non_strict continuing_with_errors=%d",
                        len(verification_result["errors"])
                    )
            else:
                logger.info(
                    "contract_verification_passed verified=%d warnings=%d",
                    len(verification_result["verified_contracts"]),
                    len(verification_result.get("warnings", []))
                )

                for warning in verification_result.get("warnings", [])[:5]:
                    logger.warning("contract_warning: %s", warning)

            # Validate construction
            if not hasattr(method_executor, 'execute'):
                if self._strict:
                    raise ExecutorConstructionError("MethodExecutor missing 'execute' method")
                logger.warning("method_executor_validation_warning missing_execute")

            self._method_executor = method_executor

            logger.info(
                "method_executor_built_successfully "
                "dispensaries=%d special_routes=%d signal_registry=injected",
                len(class_registry),
                special_routes,
            )

        except Exception as e:
            if isinstance(e, ExecutorConstructionError):
                raise
            raise ExecutorConstructionError(f"Failed to build method executor: {e}") from e

    def _load_validation_constants(self) -> dict[str, Any]:
        """Load Phase 1 validation constants (hard contracts).

        These constants are injected into Orchestrator for Phase 1 validation:
        - P01_EXPECTED_CHUNK_COUNT = 60
        - P02_MIN_TABLE_COUNT = 5
        - etc.

        Returns:
            dict[str, Any]: Validation constants.
        """
        if self._validation_constants is not None:
            logger.info("validation_constants_using_provided count=%d", len(self._validation_constants))
            return self._validation_constants

        if VALIDATION_CONSTANTS_AVAILABLE:
            try:
                raw_constants = (
                    load_validation_constants()
                    if callable(load_validation_constants)
                    else PHASE1_VALIDATION_CONSTANTS
                )
                if not isinstance(raw_constants, Mapping):
                    raise TypeError(
                        f"Validation constants must be a mapping, got {type(raw_constants)!r}"
                    )

                constants = dict(raw_constants)
                logger.info("validation_constants_loaded_from_config count=%d", len(constants))
                return constants
            except Exception:
                logger.error("validation_constants_load_failed using_defaults", exc_info=True)

        # Default validation constants
        default_constants = {
            "P01_EXPECTED_CHUNK_COUNT": 60,
            "P01_MIN_CHUNK_LENGTH": 100,
            "P01_MAX_CHUNK_LENGTH": 2000,
            "P02_MIN_TABLE_COUNT": 5,
            "P02_MAX_TABLES_PER_DOCUMENT": 100,
        }

        logger.warning(
            "validation_constants_using_defaults count=%d constants_module_unavailable",
            len(default_constants),
        )

        return default_constants

    def _get_executor_config(self) -> ExecutorConfig:
        """Get or create ExecutorConfig."""
        if self._executor_config is not None:
            return self._executor_config
        return ExecutorConfig.default()

    def _build_orchestrator(
        self,
        executor_config: ExecutorConfig,
        validation_constants: dict[str, Any],
    ) -> Orchestrator:
        """Build Orchestrator with full dependency injection.

        CRITICAL: Orchestrator receives:
        1. questionnaire: CanonicalQuestionnaire (NOT file path)
        2. method_executor: MethodExecutor
        3. executor_config: ExecutorConfig
        4. validation_constants: dict (Phase 1 hard contracts)

        Args:
            executor_config: ExecutorConfig instance.
            validation_constants: Phase 1 validation constants.
        Returns:
            Orchestrator: Fully configured orchestrator.

        Raises:
            ExecutorConstructionError: If orchestrator construction fails.
        """
        if self._canonical_questionnaire is None:
            raise ExecutorConstructionError("Cannot build orchestrator: questionnaire not loaded")
        if self._method_executor is None:
            raise ExecutorConstructionError("Cannot build orchestrator: method executor not built")

        logger.info("orchestrator_building_start")

        try:
            # Build orchestrator with FULL dependency injection
            # Local import to avoid circular dependency
            from orchestration.orchestrator import Orchestrator
            orchestrator = Orchestrator(
                questionnaire=self._canonical_questionnaire,  # DI: inject questionnaire object
                method_executor=self._method_executor,  # DI: inject method executor
                executor_config=executor_config,  # DI: inject config
                validation_constants=validation_constants,  # DI: inject Phase 1 contracts
                signal_registry=self._signal_registry,  # DI: inject signal registry
            )

            logger.info("orchestrator_built_successfully")

            return orchestrator

        except Exception as e:
            raise ExecutorConstructionError(f"Failed to build orchestrator: {e}") from e

    def _build_core_module_factory(self) -> Any | None:
        """Build CoreModuleFactory if available."""
        if not CORE_MODULE_FACTORY_AVAILABLE:
            return None

        try:
            factory = CoreModuleFactory()
            logger.info("core_module_factory_built")
            return factory
        except Exception:
            logger.error("core_module_factory_construction_error", exc_info=True)
            return None

    def _compute_questionnaire_hash(self) -> str:
        """Compute SHA-256 hash of loaded questionnaire."""
        if self._canonical_questionnaire is None:
            return ""
        return self._compute_questionnaire_hash_from_instance(self._canonical_questionnaire)

    @staticmethod
    def _compute_questionnaire_hash_from_instance(questionnaire: CanonicalQuestionnaire) -> str:
        """Compute deterministic SHA-256 hash of questionnaire content."""
        try:
            # Try to get JSON representation if available
            if hasattr(questionnaire, 'to_dict'):
                content = json.dumps(questionnaire.to_dict(), sort_keys=True)
            elif hasattr(questionnaire, '__dict__'):
                content = json.dumps(questionnaire.__dict__, sort_keys=True, default=str)
            else:
                content = str(questionnaire)

            return hashlib.sha256(content.encode('utf-8')).hexdigest()

        except Exception as e:
            logger.warning("questionnaire_hash_computation_degraded error=%s", str(e))
            # Fallback to simple string hash
            return hashlib.sha256(str(questionnaire).encode('utf-8')).hexdigest()

    def create_executor_instance(
        self,
        executor_class: type,
        policy_area_id: str,
        **extra_kwargs: Any,
    ) -> Any:
        """Create BaseExecutor instance with EnrichedSignalPack injected.

        This method is called for each of the ~30 BaseExecutor classes.
        Each executor receives its specific EnrichedSignalPack, NOT the full registry.

        Args:
            executor_class: BaseExecutor subclass to instantiate.
            policy_area_id: Policy area identifier for signal pack selection.
            **extra_kwargs: Additional kwargs to pass to constructor.

        Returns:
            BaseExecutor instance with dependencies injected.

        Raises:
            ExecutorConstructionError: If executor instantiation fails.
        """
        if self._method_executor is None:
            raise ExecutorConstructionError(
                "Cannot create executor: method executor not built"
            )

        # Get enriched signal pack for this policy area
        enriched_pack = self._enriched_packs.get(policy_area_id)

        if enriched_pack is None and self._enable_intelligence:
            logger.warning(
                "executor_creation_warning no_enriched_pack policy_area=%s executor=%s",
                policy_area_id,
                executor_class.__name__,
            )

        try:
            # Inject dependencies into executor
            executor_instance = executor_class(
                method_executor=self._method_executor,  # DI: inject method executor
                signal_registry=self._signal_registry,  # DI: inject signal registry
                config=self._get_executor_config(),  # DI: inject config
                questionnaire_provider=self._canonical_questionnaire,  # DI: inject questionnaire
                enriched_pack=enriched_pack,  # DI: inject enriched signal pack (specific to policy area)
                **extra_kwargs,
            )

            logger.debug(
                "executor_instance_created executor=%s policy_area=%s",
                executor_class.__name__,
                policy_area_id,
            )

            return executor_instance

        except Exception as e:
            raise ExecutorConstructionError(
                f"Failed to create executor {executor_class.__name__}: {e}"
            ) from e


# ============================================================================
# Convenience Functions
# ============================================================================


def create_analysis_pipeline(
    questionnaire_path: str | None = None,
    expected_hash: str | None = None,
    seed: int | None = None,
) -> ProcessorBundle:
    """Convenience function to create complete analysis pipeline.

    This is the RECOMMENDED entry point for most use cases.

    Args:
        questionnaire_path: Path to canonical questionnaire JSON.
        expected_hash: Expected SHA-256 hash for integrity check.
        seed: Seed for reproducibility.

    Returns:
        ProcessorBundle with Orchestrator ready to use.
    """
    factory = AnalysisPipelineFactory(
        questionnaire_path=questionnaire_path,
        expected_questionnaire_hash=expected_hash,
        seed_for_determinism=seed,
        enable_intelligence_layer=True,
        strict_validation=True,
    )
    return factory.create_orchestrator()


def create_minimal_pipeline(
    questionnaire_path: str | None = None,
) -> ProcessorBundle:
    """Create minimal pipeline without intelligence layer.

    Useful for testing or when enriched signals are not needed.

    Args:
        questionnaire_path: Path to canonical questionnaire JSON.

    Returns:
        ProcessorBundle with basic dependencies only.
    """
    factory = AnalysisPipelineFactory(
        questionnaire_path=questionnaire_path,
        enable_intelligence_layer=False,
        strict_validation=False,
    )
    return factory.create_orchestrator()


# Alias for backward compatibility with Phase 2 executors
build_processor = create_analysis_pipeline


# ============================================================================
# Validation and Diagnostics
# ============================================================================


def validate_factory_singleton() -> dict[str, Any]:
    """Validate that load_questionnaire() was called exactly once.

    Returns:
        dict with validation results.
    """
    return {
        "questionnaire_loaded": AnalysisPipelineFactory._questionnaire_loaded,
        "questionnaire_instance_exists": AnalysisPipelineFactory._questionnaire_instance is not None,
        "singleton_pattern_valid": (
            AnalysisPipelineFactory._questionnaire_loaded and
            AnalysisPipelineFactory._questionnaire_instance is not None
        ),
    }


def validate_bundle(bundle: ProcessorBundle) -> dict[str, Any]:
    """Validate bundle integrity and return diagnostics."""
    diagnostics = {
        "valid": True,
        "errors": [],
        "warnings": [],
        "components": {},
        "metrics": {},
    }

    # Validate orchestrator
    if bundle.orchestrator is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("orchestrator is None")
    else:
        diagnostics["components"]["orchestrator"] = "present"

    # Validate method executor
    if bundle.method_executor is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("method_executor is None")
    else:
        diagnostics["components"]["method_executor"] = "present"
        if hasattr(bundle.method_executor, 'arg_router'):
            router = bundle.method_executor.arg_router
            if hasattr(router, 'get_special_route_coverage'):
                diagnostics["metrics"]["special_routes"] = router.get_special_route_coverage()

    # Validate questionnaire
    if bundle.questionnaire is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("questionnaire is None")
    else:
        diagnostics["components"]["questionnaire"] = "present"
        if hasattr(bundle.questionnaire, 'questions'):
            diagnostics["metrics"]["question_count"] = len(bundle.questionnaire.questions)

    # Validate signal registry
    if bundle.signal_registry is None:
        diagnostics["valid"] = False
        diagnostics["errors"].append("signal_registry is None")
    else:
        diagnostics["components"]["signal_registry"] = "present"
        if hasattr(bundle.signal_registry, 'get_all_policy_areas'):
            diagnostics["metrics"]["policy_areas"] = len(bundle.signal_registry.get_all_policy_areas())

    # Validate enriched packs
    diagnostics["components"]["enriched_packs"] = len(bundle.enriched_signal_packs)
    diagnostics["metrics"]["enriched_pack_count"] = len(bundle.enriched_signal_packs)

    # Validate validation constants
    diagnostics["components"]["validation_constants"] = len(bundle.validation_constants)
    diagnostics["metrics"]["validation_constant_count"] = len(bundle.validation_constants)

    # Validate seed registry
    if not bundle.seed_registry_initialized:
        diagnostics["warnings"].append("SeedRegistry not initialized - determinism not guaranteed")

    # Check factory instantiation
    if not bundle.provenance.get("factory_instantiation_confirmed"):
        diagnostics["errors"].append("Bundle not created via AnalysisPipelineFactory")
        diagnostics["valid"] = False

    return diagnostics


def get_bundle_info(bundle: ProcessorBundle) -> dict[str, Any]:
    """Get human-readable information about bundle."""
    return {
        "construction_time": bundle.provenance.get("construction_timestamp_utc"),
        "canonical_hash": bundle.provenance.get("canonical_sha256", "")[:16],
        "policy_areas": sorted(bundle.enriched_signal_packs.keys()),
        "policy_area_count": len(bundle.enriched_signal_packs),
        "intelligence_layer": bundle.provenance.get("intelligence_layer_enabled"),
        "validation_constants": len(bundle.validation_constants),
        "construction_duration": bundle.provenance.get("construction_duration_seconds"),
        "seed_initialized": bundle.seed_registry_initialized,
        "factory_class": bundle.provenance.get("factory_class"),
    }


# ============================================================================
# Module-level Checks
# ============================================================================


def check_legacy_signal_loader_deleted() -> dict[str, Any]:
    """Check that signal_loader.py has been deleted.

    Returns:
        dict with check results.
    """
    try:
        import cross_cutting_infrastructure.irrigation_using_signals.SISAS.signal_loader
        return {
            "legacy_loader_deleted": False,
            "error": "signal_loader.py still exists - must be deleted per architecture requirements",
        }
    except ImportError:
        return {
            "legacy_loader_deleted": True,
            "message": "signal_loader.py correctly deleted - no legacy signal loading",
        }


def verify_single_questionnaire_load_point() -> dict[str, Any]:
    """Verify that only AnalysisPipelineFactory calls load_questionnaire().

    This requires manual code search but provides guidance.

    Returns:
        dict with verification instructions.
    """
    return {
        "verification_required": True,
        "search_command": "grep -r 'load_questionnaire(' --exclude-dir=__pycache__ --exclude='*.pyc'",
        "expected_result": "Should ONLY appear in: factory.py (AnalysisPipelineFactory._load_canonical_questionnaire)",
        "instructions": (
            "1. Run grep command above\n"
            "2. Verify ONLY factory.py calls load_questionnaire()\n"
            "3. Remove any other calls found\n"
            "4. Update tests to use AnalysisPipelineFactory"
        ),
    }


def get_method_dispensary_info() -> dict[str, Any]:
    """Get information about the method dispensary pattern.

    Returns detailed statistics about:
    - Which monolith classes serve as dispensaries
    - How many methods each dispensary provides
    - Which executors use which dispensaries
    - Method reuse patterns

    Returns:
        dict with dispensary statistics and usage patterns.
    """


    class_paths = get_class_paths()

    # Load executorâ\206\222methods mapping
    try:
        import json
        from pathlib import Path
        executors_methods_path = Path(__file__).parent / "executors_methods.json"
        if executors_methods_path.exists():
            with open(executors_methods_path) as f:
                executors_methods = json.load(f)
        else:
            executors_methods = []
    except Exception:
        executors_methods = []

    # Build dispensary statistics
    dispensaries = {}
    for class_name in class_paths.keys():
        dispensaries[class_name] = {
            "module": class_paths[class_name],
            "methods_provided": [],
            "used_by_executors": [],
            "total_usage_count": 0,
        }

    # Count method usage per dispensary
    for executor_info in executors_methods:
        executor_id = executor_info.get("executor_id")
        methods = executor_info.get("methods", [])

        for method_info in methods:
            class_name = method_info.get("class")
            method_name = method_info.get("method")
            if class_name in dispensaries:
                if method_name not in dispensaries[class_name]["methods_provided"]:
                    dispensaries[class_name]["methods_provided"].append(method_name)

                if executor_id not in dispensaries[class_name]["used_by_executors"]:
                    dispensaries[class_name]["used_by_executors"].append(executor_id)

                dispensaries[class_name]["total_usage_count"] += 1

    # Sort by usage count
    sorted_dispensaries = sorted(
        dispensaries.items(),
        key=lambda x: x[1]["total_usage_count"],
        reverse=True
    )

    # Build summary statistics
    total_methods = sum(len(d["methods_provided"]) for _, d in sorted_dispensaries)
    total_usage = sum(d["total_usage_count"] for _, d in sorted_dispensaries)

    return {
        "pattern": "method_dispensary",
        "description": "Monolith classes serve as method dispensaries for 30 executors",
        "total_dispensaries": len(dispensaries),
        "total_unique_methods": total_methods,
        "total_method_calls": total_usage,
        "avg_reuse_per_method": round(total_usage / max(total_methods, 1), 2),
        "dispensaries": {
            name: {
                "methods_count": len(info["methods_provided"]),
                "executor_count": len(info["used_by_executors"]),
                "total_calls": info["total_usage_count"],
                "reuse_factor": round(info["total_usage_count"] / max(len(info["methods_provided"]), 1), 2),
            }
            for name, info in sorted_dispensaries[:10]  # Top 10
        },
        "top_dispensaries": [
            {
                "class": name,
                "methods": len(info["methods_provided"]),
                "executors": len(info["used_by_executors"]),
                "calls": info["total_usage_count"],
            }
            for name, info in sorted_dispensaries[:5]
        ],
    }


def validate_method_dispensary_pattern() -> dict[str, Any]:
    """Validate that the method dispensary pattern is correctly implemented.

    Checks:
    1. All executor methods exist in class_registry
    2. No executor directly imports monolith classes
    3. All methods route through MethodExecutor
    4. Signal registry is injected (not globally accessed)

    Returns:
        dict with validation results.
    """


    class_paths = get_class_paths()
    validation_results = {
        "pattern_valid": True,
        "errors": [],
        "warnings": [],
        "checks": {},
    }

    # Check 1: Verify class_registry is populated
    if not class_paths:
        validation_results["pattern_valid"] = False
        validation_results["errors"].append(
            "class_registry is empty - no dispensaries registered"
        )
    else:
        validation_results["checks"]["dispensaries_registered"] = len(class_paths)

    # Check 2: Verify executor_methods.json exists
    try:
        import json
        from pathlib import Path
        executors_methods_path = Path(__file__).parent / "executors_methods.json"
        if not executors_methods_path.exists():
            validation_results["warnings"].append(
                "executors_methods.json not found - cannot validate method mappings"
            )
        else:
            with open(executors_methods_path) as f:
                executors_methods = json.load(f)
            validation_results["checks"]["executor_method_mappings"] = len(executors_methods)
    except Exception as e:
        validation_results["warnings"].append(
            f"Failed to load executors_methods.json: {e}"
        )

    # Check 3: Verify validation file exists
    try:
        validation_path = Path(__file__).parent / "executor_factory_validation.json"
        if not validation_path.exists():
            validation_results["warnings"].append(
                "executor_factory_validation.json not found - cannot validate method catalog"
            )
        else:
            with open(validation_path) as f:
                validation_data = json.load(f)
            validation_results["checks"]["method_pairs_validated"] = validation_data.get("validated_against_catalog", 0)
            validation_results["checks"]["validation_failures"] = len(validation_data.get("failures", []))
    except Exception as e:
        validation_results["warnings"].append(
            f"Failed to load executor_factory_validation.json: {e}"
        )

    return validation_results


# _validate_questionnaire_structure moved to orchestration.questionnaire_validation
# to break import cycle between factory and orchestrator.



################################################################################
# FILE: phase2_x_class_registry.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_x_class_registry.py
################################################################################

"""
Module: phase2_x_class_registry
PHASE_LABEL: Phase 2
Sequence: X

"""
"""Dynamic class registry for orchestrator method execution."""
from __future__ import annotations

from importlib import import_module
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from collections.abc import Mapping

class ClassRegistryError(RuntimeError):
    """Raised when one or more classes cannot be loaded."""

# Map of orchestrator-facing class names to their import paths.
# CORRECTED: Changed from non-existent ’farfan_core’ to actual ’methods_dispensary’ package
_CLASS_PATHS: Mapping[str, str] = {
    # Policy Processing
    "IndustrialPolicyProcessor": "methods_dispensary.policy_processor.IndustrialPolicyProcessor",
    "PolicyTextProcessor": "methods_dispensary.policy_processor.PolicyTextProcessor",
    "BayesianEvidenceScorer": "methods_dispensary.policy_processor.BayesianEvidenceScorer",

    # Contradiction Detection
    "PolicyContradictionDetector": "methods_dispensary.contradiction_deteccion.PolicyContradictionDetector",
    "TemporalLogicVerifier": "methods_dispensary.contradiction_deteccion.TemporalLogicVerifier",
    "BayesianConfidenceCalculator": "methods_dispensary.contradiction_deteccion.BayesianConfidenceCalculator",

    # Financial Analysis (derek_beach.py)
    "PDETMunicipalPlanAnalyzer": "methods_dispensary.financiero_viabilidad_tablas.PDETMunicipalPlanAnalyzer",
    "CDAFFramework": "methods_dispensary.derek_beach.CDAFFramework",
    "CausalExtractor": "methods_dispensary.derek_beach.CausalExtractor",
    "OperationalizationAuditor": "methods_dispensary.derek_beach.OperationalizationAuditor",
    "FinancialAuditor": "methods_dispensary.derek_beach.FinancialAuditor",
    "BayesianMechanismInference": "methods_dispensary.derek_beach.BayesianMechanismInference",
    "BayesianCounterfactualAuditor": "methods_dispensary.derek_beach.BayesianCounterfactualAuditor",

    # Embedding & Semantic Processing
    "BayesianNumericalAnalyzer": "methods_dispensary.embedding_policy.BayesianNumericalAnalyzer",
    "PolicyAnalysisEmbedder": "methods_dispensary.embedding_policy.PolicyAnalysisEmbedder",
    "AdvancedSemanticChunker": "methods_dispensary.embedding_policy.AdvancedSemanticChunker",
    "EmbeddingPolicyProducer": "methods_dispensary.embedding_policy.EmbeddingPolicyProducer",
    # SemanticChunker is an alias maintained for backwards compatibility.
    "SemanticChunker": "methods_dispensary.embedding_policy.AdvancedSemanticChunker",
    "SemanticProcessor": "methods_dispensary.semantic_chunking_policy.SemanticProcessor",
    "SemanticChunkingProducer": "methods_dispensary.semantic_chunking_policy.SemanticChunkingProducer",

    # Analyzer One
    "SemanticAnalyzer": "methods_dispensary.analyzer_one.SemanticAnalyzer",
    "PerformanceAnalyzer": "methods_dispensary.analyzer_one.PerformanceAnalyzer",
    "TextMiningEngine": "methods_dispensary.analyzer_one.TextMiningEngine",
    "MunicipalOntology": "methods_dispensary.analyzer_one.MunicipalOntology",

    # Teoria de Cambio
    "TeoriaCambio": "methods_dispensary.teoria_cambio.TeoriaCambio",
    "AdvancedDAGValidator": "methods_dispensary.teoria_cambio.AdvancedDAGValidator",
    "IndustrialGradeValidator": "methods_dispensary.teoria_cambio.IndustrialGradeValidator",

    # Derek Beach - Additional Classes
    "BeachEvidentialTest": "methods_dispensary.derek_beach.BeachEvidentialTest",
    "ConfigLoader": "methods_dispensary.derek_beach.ConfigLoader",
    "PDFProcessor": "methods_dispensary.derek_beach.PDFProcessor",
    "ReportingEngine": "methods_dispensary.derek_beach.ReportingEngine",
    "BayesFactorTable": "methods_dispensary.derek_beach.BayesFactorTable",
    "AdaptivePriorCalculator": "methods_dispensary.derek_beach.AdaptivePriorCalculator",
    "HierarchicalGenerativeModel": "methods_dispensary.derek_beach.HierarchicalGenerativeModel",

    # Evidence Nexus (replaced EvidenceAssembler)
    "EvidenceNexus": "canonic_phases.Phase_two.evidence_nexus.EvidenceNexus",
    "EvidenceAssembler": "canonic_phases.Phase_two.evidence_nexus.EvidenceNexus",  # Alias for backwards compatibility

    # Executors (in canonic_phases/Phase_two/executors.py)
    # D1_Q1_QuantitativeBaselineExtractor and D1_Q2_ProblemDimensioningAnalyzer removed (Legacy)

    # Additional classes that may be referenced in contracts
    "MechanismPartExtractor": "methods_dispensary.derek_beach.MechanismPartExtractor",
    "CausalInferenceSetup": "methods_dispensary.derek_beach.CausalInferenceSetup",
}

def build_class_registry() -> dict[str, type[object]]:
    """Return a mapping of class names to loaded types, validating availability.

    Classes that depend on optional dependencies (e.g., torch) are skipped
    gracefully if those dependencies are not available.
    """
    resolved: dict[str, type[object]] = {}
    missing: dict[str, str] = {}
    skipped_optional: dict[str, str] = {}

    for name, path in _CLASS_PATHS.items():
        module_name, _, class_name = path.rpartition(".")
        if not module_name:
            missing[name] = path
            continue
        try:
            module = import_module(module_name)
        except ImportError as exc:
            exc_str = str(exc)
            # Check if this is an optional dependency error
            optional_deps = [
                "torch", "tensorflow", "pyarrow", "camelot",
                "sentence_transformers", "transformers", "spacy",
                "pymc", "arviz", "dowhy", "econml"
            ]
            if any(opt_dep in exc_str for opt_dep in optional_deps):
                # Mark as skipped optional rather than missing
                skipped_optional[name] = f"{path} (optional dependency: {exc})"
            else:
                missing[name] = f"{path} (import error: {exc})"
            continue
        try:
            attr = getattr(module, class_name)
        except AttributeError:
            missing[name] = f"{path} (attribute missing)"
        else:
            if not isinstance(attr, type):
                missing[name] = f"{path} (attribute is not a class: {type(attr).__name__})"
            else:
                resolved[name] = attr

    # Log skipped optional dependencies
    if skipped_optional:
        import logging
        logger = logging.getLogger(__name__)
        logger.info(
            f"Skipped {len(skipped_optional)} optional classes due to missing dependencies: "
            f"{', '.join(skipped_optional.keys())}"
        )

    if missing:
        formatted = ", ".join(f"{name}: {reason}" for name, reason in missing.items())
        raise ClassRegistryError(f"Failed to load orchestrator classes: {formatted}")
    return resolved

def get_class_paths() -> Mapping[str, str]:
    """Expose the raw class path mapping for diagnostics."""
    return _CLASS_PATHS



################################################################################
# FILE: phase2_y_schema_validation.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_y_schema_validation.py
################################################################################

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

PHASE_LABEL: Phase 2
PHASE_COMPONENT: Schema Validation (Phase 6 validation logic)
PHASE_ROLE: Validates schema compatibility between questions and chunks during task construction

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 â\206\222 Phase 6.2 â\206\222 Phase 6.3 â\206\222 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}]"
        )
```

Phase 2 Source Code

```
    # 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",
]



################################################################################
# FILE: phase2_z_task_executor.py
# PATH: src/farfan_pipeline/phases/Phase_two/phase2_z_task_executor.py
################################################################################

"""
Module: phase2_z_task_executor
PHASE_LABEL: Phase 2
Sequence: Z

"""
Module: src.canonic_phases.phase_2.phase2_e_task_executor
Purpose: Phase 2.2 Task Execution - Execute 300 tasks from ExecutionPlan
Owner: phase2_orchestration
Lifecycle: ACTIVE
Version: 1.0.0
Effective-Date: 2025-12-19
Python-Version: 3.12+

Contracts-Enforced:
    - ExecutionContract: All 300 tasks from ExecutionPlan execute successfully
    - DeterminismContract: Same task inputs produce identical outputs
    - ProvenanceContract: Each output traces to originating task
    - CalibrationContract: Optional method calibration before execution

Determinism:
    Seed-Strategy: INHERITED from ExecutionPlan correlation_id
    State-Management: Executor caches base_slot derivations, otherwise stateless

Inputs:
    - execution_plan: ExecutionPlan â\200\224 300 tasks from Phase 2.1
    - preprocessed_document: PreprocessedDocument â\200\224 60 CPP chunks
    - questionnaire_monolith: dict â\200\224 300 questions for context
    - signal_registry: SignalRegistry â\200\224 REQUIRED SISAS signal resolution
    - calibration_orchestrator: Optional â\200\224 Method calibration
    - validation_orchestrator: Optional â\200\224 Validation tracking

Outputs:
    - task_results: list[TaskResult] â\200\224 300 task execution results
    - OR raises ExecutionError

Failure-Modes:
    - TaskExecutionFailure: ExecutionError â\200\224 Task execution failed
    - QuestionLookupFailure: ValueError â\200\224 Cannot find question for task
    - ExecutorInstantiationFailure: ExecutionError â\200\224 Cannot create executor
    - CalibrationFailure: CalibrationError â\200\224 Method calibration failed

Phase 2.2 Process:
    1. Iterate over ExecutionPlan.tasks (300 tasks)
    2. For each task:
       a. Lookup question from monolith
       b. Build question_context
       c. Instantiate/reuse DynamicContractExecutor
       d. Execute task with executor
       e. Collect result
    3. Return list of 300 TaskResult objects
"""
from __future__ import annotations

from dataclasses import dataclass, field
from typing import Any, Final
import logging
import threading
from datetime import datetime, timezone

from .phase2_d_irrigation_orchestrator import ExecutionPlan, ExecutableTask

logger: Final = logging.getLogger(__name__)

# === DATA STRUCTURES ===

@dataclass(frozen=True, slots=True)
class TaskResult:
    """
    Result of executing a single task.
    Invariants:
        - task_id matches originating ExecutableTask
        - success indicates execution completed
        - output contains executor results
    """
    task_id: str
    question_id: str
    question_global: int
    policy_area_id: str
    dimension_id: str
    chunk_id: str
    success: bool
    output: dict[str, Any]
    error: str | None = None
    execution_time_ms: float | None = None
    metadata: dict[str, Any] = field(default_factory=dict)


@dataclass(frozen=True, slots=True)
class QuestionContext:
    """
    Context for a question ready for executor dispatch.

    Contains all data needed to execute a question against a chunk.
    """
    question_id: str
    question_global: int
    question_text: str
    policy_area_id: str
    dimension_id: str
    chunk_id: str
    chunk_text: str
    patterns: list[dict[str, Any]]
    signals: dict[str, Any]
    expected_elements: list[dict[str, Any]]
    method_sets: list[str]
    correlation_id: str
    metadata: dict[str, Any] = field(default_factory=dict)


# === EXCEPTION TAXONOMY ===

@dataclass
class ExecutionError(Exception):
    """Raised when Phase 2.2 task execution fails."""
    error_code: str
    message: str
    task_id: str | None = None
    details: dict[str, Any] = field(default_factory=dict)

    def __str__(self) -> str:
        if self.task_id:
            return f"[{self.error_code}] Task {self.task_id}: {self.message}"
        return f"[{self.error_code}] {self.message}"


@dataclass
class CalibrationError(Exception):
    """Raised when method calibration fails."""
    error_code: str
    message: str
    method_name: str | None = None


# === DYNAMIC CONTRACT EXECUTOR ===

class DynamicContractExecutor:
    """
    Executor for the 300-contract model with automatic base_slot derivation.

    Derives base_slot from question_id using the formula:
    - slot_index = (q_number - 1) % 30
    - dimension = (slot_index // 5) + 1
    - question_in_dimension = (slot_index % 5) + 1
    - base_slot = f"D{dimension}-Q{question_in_dimension}"

    Caches derivations in _question_to_base_slot_cache for performance.

    SUCCESS_CRITERIA:
        - Correct base_slot derivation for all Q001-Q300
        - Successful method execution for all tasks
        - Output format compatible with carver input

    FAILURE_MODES:
        - InvalidQuestionID: Cannot parse question_id
        - BaseSlotDerivationFailure: Formula produces invalid slot
        - MethodExecutionFailure: Executor method fails

    VERIFICATION_STRATEGY:
        - test_phase2_task_executor.py
    """

    # Class-level cache for base_slot derivations
    _question_to_base_slot_cache: dict[str, str] = {}
    _cache_lock = threading.Lock()
```

Phase 2 Source Code

```
    def __init__(
        self,
        question_id: str,
        calibration_orchestrator: Any | None = None,
        validation_orchestrator: Any | None = None,
    ) -> None:
        """
        Initialize DynamicContractExecutor for a specific question.

        Args:
            question_id: Question identifier (e.g., "Q001", "Q150")
            calibration_orchestrator: Optional calibration support
            validation_orchestrator: Optional validation tracking
        """
        self.question_id = question_id
        self.calibration_orchestrator = calibration_orchestrator
        self.validation_orchestrator = validation_orchestrator

        # Derive and cache base_slot
        self.base_slot = self._derive_base_slot(question_id)

        logger.info(
            "DynamicContractExecutor initialized",
            extra={
                "question_id": question_id,
                "base_slot": self.base_slot,
            }
        )

    @classmethod
    def _derive_base_slot(cls, question_id: str) -> str:
        """
        Derive base_slot from question_id with thread-safe caching.

        Formula:
        - Extract question number (Q001 -> 1, Q150 -> 150)
        - slot_index = (q_number - 1) % 30
        - dimension = (slot_index // 5) + 1
        - question_in_dimension = (slot_index % 5) + 1
        - base_slot = f"D{dimension}-Q{question_in_dimension}"

        Examples:
        - Q001 -> slot_index=0 -> D1-Q1
        - Q006 -> slot_index=5 -> D2-Q1
        - Q030 -> slot_index=29 -> D6-Q5
        - Q031 -> slot_index=0 -> D1-Q1 (wraps)
        """
        # Thread-safe cache access
        with cls._cache_lock:
            if question_id in cls._question_to_base_slot_cache:
                return cls._question_to_base_slot_cache[question_id]

        # Parse question number
        try:
            if not question_id.startswith("Q"):
                raise ValueError(f"Invalid question_id format: {question_id}")
            q_number = int(question_id[1:])
        except (ValueError, IndexError) as e:
            raise ValueError(f"Cannot parse question_id: {question_id}") from e

        # Derive slot_index
        slot_index = (q_number - 1) % 30

        # Derive dimension and question_in_dimension
        dimension = (slot_index // 5) + 1
        question_in_dimension = (slot_index % 5) + 1

        # Build base_slot
        base_slot = f"D{dimension}-Q{question_in_dimension}"

        # Thread-safe cache write
        with cls._cache_lock:
            cls._question_to_base_slot_cache[question_id] = base_slot

        return base_slot

    def execute(self, question_context: QuestionContext) -> dict[str, Any]:
        """
        Execute task with question context.

        Args:
            question_context: Context with all data for execution

        Returns:
            Execution result dictionary

        Raises:
            ExecutionError: If execution fails
        """
        start_time = datetime.now(timezone.utc)

        try:
            # Build method context
            method_context = self._build_method_context(question_context)

            # Execute methods (simplified - actual implementation would call real executors)
            output = self._execute_methods(method_context, question_context)

            # Track execution time
            end_time = datetime.now(timezone.utc)
            execution_time_ms = (end_time - start_time).total_seconds() * 1000

            logger.info(
                "Task execution successful",
                extra={
                    "question_id": question_context.question_id,
                    "base_slot": self.base_slot,
                    "execution_time_ms": execution_time_ms,
                }
            )

            return {
                "question_id": question_context.question_id,
                "base_slot": self.base_slot,
                "output": output,
                "execution_time_ms": execution_time_ms,
                "success": True,
            }

        except Exception as e:
            logger.error(
                "Task execution failed",
                extra={
                    "question_id": question_context.question_id,
                    "base_slot": self.base_slot,
                    "error": str(e),
                }
            )
            raise ExecutionError(
                error_code="E2007",
                message=f"Task execution failed: {str(e)}",
                task_id=question_context.question_id,
                details={"base_slot": self.base_slot, "error": str(e)}
            ) from e

    def _build_method_context(
        self, question_context: QuestionContext
    ) -> dict[str, Any]:
        """Build context dictionary for method execution."""
        return {
            "question_id": question_context.question_id,
            "question_global": question_context.question_global,
            "question_text": question_context.question_text,
            "policy_area_id": question_context.policy_area_id,
            "dimension_id": question_context.dimension_id,
            "base_slot": self.base_slot,
            "chunk_id": question_context.chunk_id,
            "chunk_text": question_context.chunk_text,
            "patterns": question_context.patterns,
            "signals": question_context.signals,
            "expected_elements": question_context.expected_elements,
            "method_sets": question_context.method_sets,
            "correlation_id": question_context.correlation_id,
        }

    def _execute_methods(
        self, method_context: dict, question_context: QuestionContext
    ) -> dict[str, Any]:
        """
        Execute methods for this question.

        OPERATIONAL INTEGRATION:
        This method integrates with the existing MethodRegistry infrastructure:

        1. MethodRegistry implements lazy loading with 300s TTL cache
        2. 40+ method classes mapped in class_registry._CLASS_PATHS:
           - TextMiningEngine, CausalExtractor, FinancialAuditor,
           - BayesianNumericalAnalyzer, PolicyAnalysisEmbedder, etc.
        3. Integration flow:
           - Read method_binding.methods[] from contract v3
           - Call MethodRegistry.get_method(class_name, method_name)
           - Instantiate class under demand from methods_dispensary/*
           - Execute with arguments validated by ExtendedArgRouter
        4. CalibrationPolicy (from calibration_policy.py) weights methods
        5. Thread-safe with threading.Lock

        Current Implementation:
        - Simplified execution for canonical Phase 2 pipeline
        - Full MethodRegistry integration available via orchestrator
        - See: farfan_pipeline/orchestration/method_registry.py
        - See: farfan_pipeline/phases/Phase_two/calibration_policy.py
        """
        # Simplified execution - full integration via orchestrator’s MethodRegistry
        return {
            "method_outputs": {},
            "patterns_matched": len(question_context.patterns),
            "signals_resolved": len(question_context.signals),
            "expected_elements": question_context.expected_elements,
        }


# === TASK EXECUTOR ===
```

Phase 2 Source Code

```python
class TaskExecutor:
    """
    Phase 2.2 - Execute 300 tasks from ExecutionPlan.

    Iterates over ExecutionPlan.tasks, executes each task with
    DynamicContractExecutor, and collects results.

    SUCCESS_CRITERIA:
        - All 300 tasks execute successfully
        - Each result traces to originating task
        - Results compatible with Carver input

    FAILURE_MODES:
        - TaskExecutionFailure: Individual task fails
        - QuestionLookupFailure: Cannot find question
        - ExecutorFailure: Executor instantiation fails

    TERMINATION_CONDITION:
        - All 300 tasks processed
        - Returns list of 300 TaskResult objects

    VERIFICATION_STRATEGY:
        - test_phase2_task_executor.py
    """

    def __init__(
        self,
        questionnaire_monolith: dict[str, Any],
        preprocessed_document: Any,
        signal_registry: Any,
        calibration_orchestrator: Any | None = None,
        validation_orchestrator: Any | None = None,
    ) -> None:
        """
        Initialize TaskExecutor.

        Args:
            questionnaire_monolith: 300 questions
            preprocessed_document: 60 CPP chunks
            signal_registry: REQUIRED SISAS signal resolution (must be initialized in Phase 0)
            calibration_orchestrator: Optional calibration
            validation_orchestrator: Optional validation tracking

        Raises:
            ValueError: If signal_registry is None
        """
        # Validate SignalRegistry is provided
        if signal_registry is None:
            raise ValueError(
                "SignalRegistry is required for Phase 2.2. "
                "Must be initialized in Phase 0."
            )

        self.questionnaire_monolith = questionnaire_monolith
        self.preprocessed_document = preprocessed_document
        self.signal_registry = signal_registry
        self.calibration_orchestrator = calibration_orchestrator
        self.validation_orchestrator = validation_orchestrator

        # Build question lookup index
        self._question_index = self._build_question_index()

        # Executor cache
        self._executor_cache: dict[str, DynamicContractExecutor] = {}

    def _build_question_index(self) -> dict[str, dict[str, Any]]:
        """Build index of questions by question_id."""
        index: dict[str, dict[str, Any]] = {}

        blocks = self.questionnaire_monolith.get("blocks", [])
        for block in blocks:
            if block.get("block_type") == "micro_questions":
                for question in block.get("micro_questions", []):
                    question_id = question.get("question_id")
                    if question_id:
                        index[question_id] = question

        return index

    def execute_plan(self, execution_plan: ExecutionPlan) -> list[TaskResult]:
        """
        Execute all tasks in ExecutionPlan.

        Args:
            execution_plan: Plan with 300 tasks from Phase 2.1

        Returns:
            List of 300 TaskResult objects

        Raises:
            ExecutionError: If execution fails
        """
        results: list[TaskResult] = []

        logger.info(
            "Starting task execution",
            extra={
                "plan_id": execution_plan.plan_id,
                "task_count": len(execution_plan.tasks),
                "correlation_id": execution_plan.correlation_id,
            }
        )

        for i, task in enumerate(execution_plan.tasks):
            try:
                result = self._execute_task(task)
                results.append(result)

                if (i + 1) % 50 == 0:
                    logger.info(
                        f"Progress: {i + 1}/{len(execution_plan.tasks)} tasks completed"
                    )

            except Exception as e:
                logger.error(
                    "Task execution failed",
                    extra={
                        "task_id": task.task_id,
                        "question_id": task.question_id,
                        "error": str(e),
                    }
                )

                # Create failure result
                result = TaskResult(
                    task_id=task.task_id,
                    question_id=task.question_id,
                    question_global=task.question_global,
                    policy_area_id=task.policy_area_id,
                    dimension_id=task.dimension_id,
                    chunk_id=task.chunk_id,
                    success=False,
                    output={},
                    error=str(e),
                )
                results.append(result)

        logger.info(
            "Task execution complete",
            extra={
                "plan_id": execution_plan.plan_id,
                "total_tasks": len(results),
                "successful": sum(1 for r in results if r.success),
                "failed": sum(1 for r in results if not r.success),
            }
        )

        return results

    def _execute_task(self, task: ExecutableTask) -> TaskResult:
        """Execute single task."""
        start_time = datetime.now(timezone.utc)

        # Lookup question from monolith
        question = self._lookup_question(task)

        # Build question context
        question_context = self._build_question_context(task, question)

        # Get or create executor
        executor = self._get_executor(task.question_id)

        # Execute task
        output = executor.execute(question_context)

        # Calculate execution time
        end_time = datetime.now(timezone.utc)
        execution_time_ms = (end_time - start_time).total_seconds() * 1000

        return TaskResult(
            task_id=task.task_id,
            question_id=task.question_id,
            question_global=task.question_global,
            policy_area_id=task.policy_area_id,
            dimension_id=task.dimension_id,
            chunk_id=task.chunk_id,
            success=True,
            output=output,
            execution_time_ms=execution_time_ms,
            metadata={
                "base_slot": output.get("base_slot"),
                "correlation_id": task.correlation_id,
            }
        )

    def _lookup_question(self, task: ExecutableTask) -> dict[str, Any]:
        """Lookup question from monolith by question_id."""
        question = self._question_index.get(task.question_id)
        if not question:
            raise ValueError(
                f"Question not found in monolith: {task.question_id}"
            )
        return question

    def _build_question_context(
        self, task: ExecutableTask, question: dict
    ) -> QuestionContext:
        """Build QuestionContext from task and question."""
        return QuestionContext(
            question_id=task.question_id,
            question_global=task.question_global,
            question_text=task.question_text,
            policy_area_id=task.policy_area_id,
            dimension_id=task.dimension_id,
            chunk_id=task.chunk_id,
            chunk_text=task.chunk_text,
            patterns=task.patterns,
            signals=task.signals,
            expected_elements=task.expected_elements,
            method_sets=question.get("method_sets", []),
            correlation_id=task.correlation_id,
            metadata=task.metadata,
        )

    def _get_executor(self, question_id: str) -> DynamicContractExecutor:
        """Get or create executor for question_id (with caching)."""
        if question_id not in self._executor_cache:
            self._executor_cache[question_id] = DynamicContractExecutor(
                question_id=question_id,
                calibration_orchestrator=self.calibration_orchestrator,
                validation_orchestrator=self.validation_orchestrator,
            )
        return self._executor_cache[question_id]


# === PUBLIC API ===

def execute_tasks(
    execution_plan: ExecutionPlan,
    questionnaire_monolith: dict[str, Any],
    preprocessed_document: Any,
    signal_registry: Any | None = None,
    calibration_orchestrator: Any | None = None,
    validation_orchestrator: Any | None = None,
) -> list[TaskResult]:
    """
    Public API for executing tasks from ExecutionPlan.

    Args:
        execution_plan: Plan with 300 tasks from Phase 2.1
        questionnaire_monolith: 300 questions
        preprocessed_document: 60 CPP chunks
        signal_registry: SISAS signal resolution
        calibration_orchestrator: Optional calibration
        validation_orchestrator: Optional validation tracking

    Returns:
        List of 300 TaskResult objects

    Raises:
        ExecutionError: If execution fails
    """
    executor = TaskExecutor(
        questionnaire_monolith=questionnaire_monolith,
        preprocessed_document=preprocessed_document,
        signal_registry=signal_registry,
        calibration_orchestrator=calibration_orchestrator,
        validation_orchestrator=validation_orchestrator,
    )
    return executor.execute_plan(execution_plan)
```